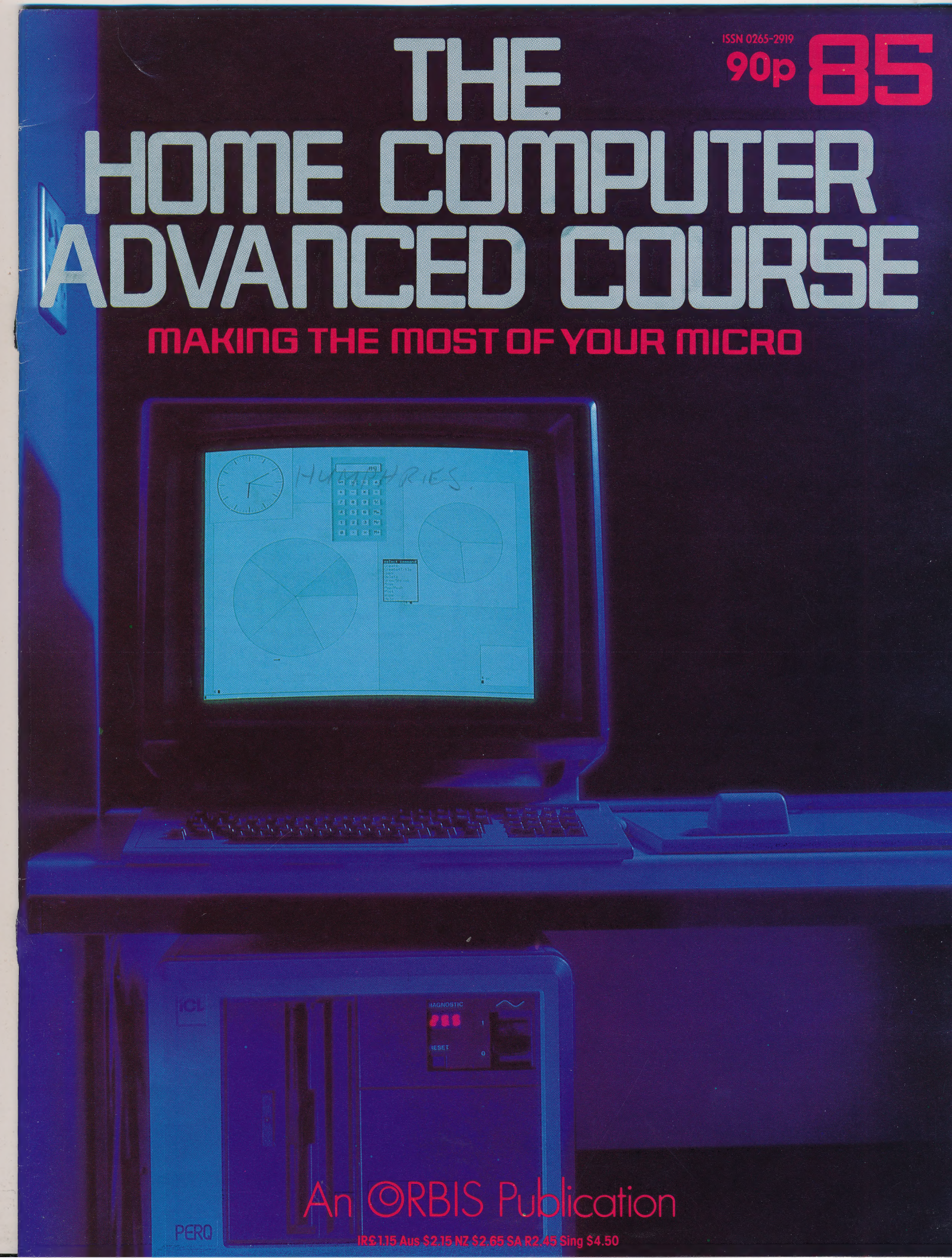
ISSN 0265-2919

90p 85

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



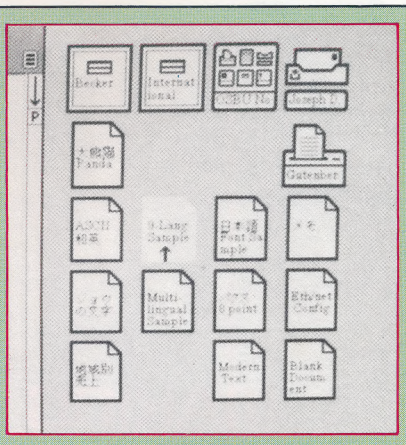
An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

Next Week

- **As a follow-up to our series on computers and gambling, we begin a project to design a program that simulates a game of blackjack.**
- **WordStar is undoubtedly the most popular word processing package now available. We look at how it works.**
- **Having taken a look at the development of WIMP systems in general, we begin our detailed examination of the GEM operating environment.**



QUIZ

1) Can you name the four pin groups on an eight-bit processor?
2) When using the 68000 Assembly language, what is the purpose of extra letters placed after an ADD instruction?
3) What is a 'blitter' and what is its function?
4) What is a word processing language?

Answers To Last Week's Quiz

1) The data division of a COBOL program is used to assign a length to record fields and to organise them into hierarchical structures where necessary. Variables required within the program are also initialised here.

2) A 'stream' is an area of an operating system which manages the flow of information to and from a particular I/O device. Streams are necessary to prevent data intended for different peripherals getting mixed up.

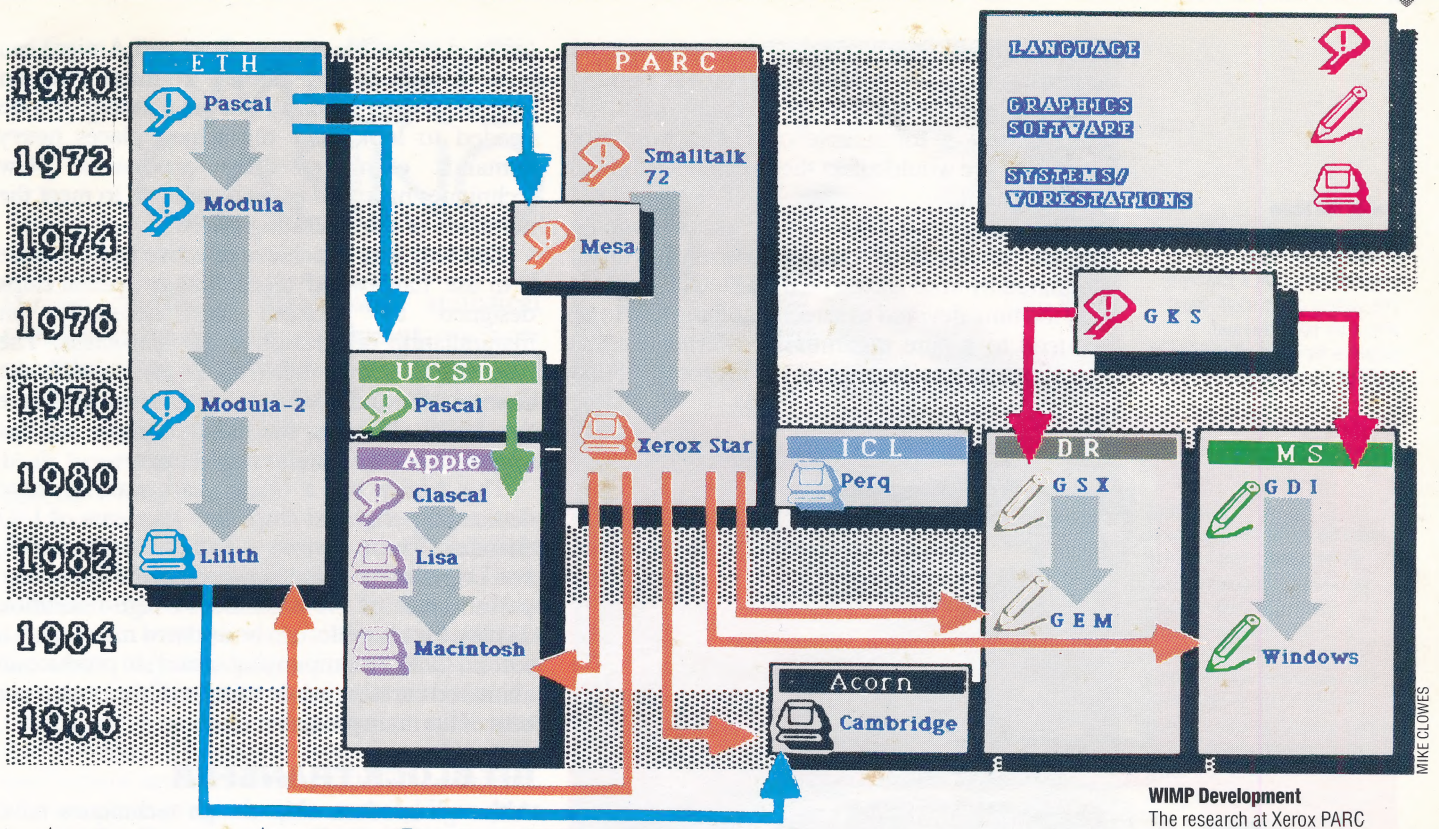
3) Because the 'objects' in object-oriented programming contain all the necessary information required to process incoming data, the message sender does not need to 'know' how the task is to be performed. Thus an 'object' is discrete and essentially modular in nature.

4) A microprogrammable computer is used by microchip manufacturers to design the electronic steps necessary to program a processor's instruction set.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Chris Honey, Chris Laing, Mike Curtis, Steve Cooke, Steve Colwill, Steve Malone, Nigel Cross, Richard Forsythe, David Fensome, Bobby Pickering; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH — SHOT AT I.C.L., BRIDGE HOUSE, LONDON



WIMP Development
The research at Xerox PARC influenced many different areas of language and hardware development. Niklaus Wirth spent a year at PARC and the results can be clearly seen here in the subsequent development of PASCAL and Modula-2. Many of the earlier products are still available. For example, GKS, the Graphics Kernel System, which brings machine independence to graphics programming, has grown into a complex (and expensive) application, though it remains limited to CAD-CAM users and research. At the other end of the scale, Digital Research's GSX, an operating system extension, has appeared on the Amstrad PCW 8256 word processor, bundled with CP/M. The following abbreviations are used:

ETH — ElektroTechnische Hochschule (Professor Niklaus Wirth)
PARC — Xerox Palo Alto Research Center
UCSD — University of California at San Diego
GKS — Graphics Kernel System (standardised in 1977)
DR — Digital Research Ltd
GSX — Graphics System Extension (DR)
GEM — Graphics Environment Manager (DR)
MS — Microsoft Corporation
GDI — Graphics Device Interface (MS)

ALL ABOUT THE BLITTERS

Our discussion of the background to the development of Digital Research's Graphics Environment Manager (GEM) continues with a look at some early, commercially available WIMP systems. We also examine the hardware problems that had to be overcome to produce such machines.

SMALLTALK and its successors GEM, Macintosh, and MS-Windows all make huge demands on hardware, and, as we discussed in the last instalment (see page 1670), it was the researchers at Xerox PARC who were given the task of developing machines that were capable of running the new systems.

The Xerox Star was launched in 1980 and drew largely on research by Xerox into the MMI (man-machine interface) and the object-oriented SMALLTALK environment. The Star used a three-button mouse, with windows and icons on a high resolution bit-mapped screen. This radical approach to computing, together with a lack of available software and the cost of the machine meant that the Star was not a commercial success.

It did, however, serve to make people aware of the possibilities of WIMPs, and in the UK, ICL produced the Perq workstation, borrowing many of the Star's features. The Perq was targeted at the scientific and academic world, and emphasised features such as its CAD graphics and the ability to merge text with graphics. The system included a high-definition monochrome screen, windowing, menu and pointer bars, and cost around £10,000. It had a reasonable commercial success, selling over 2,500 units in the UK, although this was largely attributed to its networking compatibility with ICL mainframes. The Perq was not quite as radical as the Star — hence its success!

It was not until the release of the Apple Lisa in 1983 that WIMP technology really caught the computing industry's attention. Aggressively marketed and priced at under $10,000 (about £6,500 in the UK), the machine generated enormous interest. However, it was criticised for its slow disk access and lack of software and was commercially overshadowed by the release of the IBM PC. Although rechristened the Macintosh XL and reduced in price, it was finally discontinued in 1985.

HARDWARE REQUIREMENTS

In order to implement effective graphics for icons, menus and windowing, a high-speed bit-mapped display is essential. At the heart of GEM, or any similar graphics kernel, there must be a very powerful method of moving images across the screen. Although SMALLTALK was an interpreted rather than a compiled system, the graphics algorithms still needed to be extremely efficient.

Even without considering any graphics-based application, the job of managing the overlapping windows and keeping their changing contents updated was a formidable one. A slowness of response here would affect the 'friendliness' of the whole system.

Furthermore, the requirements of running several programs or tasks concurrently (albeit simulated by time-slicing on a single CPU) meant that the time devoted to screen handling had to be reduced to a bare minimum. Powerful modern 16/32-bit processors such as the Motorola 68000 (as used in the Macintosh, Atari and Amiga) can achieve speeds of about 2 mips (million instructions per second), but the quantity of code needed to look after the screen places heavy demands even on these processors. New techniques have had to be developed to meet the demands of the operating systems.

The only way to achieve effective performance is to use 'hard-wired' algorithms or special chips designed to perform the necessary bit manipulation on receipt of an instruction. The main CPU is then left free to get on with the computation and need not take time out to handle the VDU. Perhaps the best example of this approach is the Commodore Amiga.

The Amiga has a 68000 harnessed to three dedicated peripheral chips that take care of I/O channels. These devices — named Portia, Agnes, and Daphne — account for much of the impressive performance of the machine. A high-resolution display, for example, can be updated many times a second (with accompanying sound) to produce an animated cartoon, while using as little as nine per cent of the main processor time.

Star Of The Show
The Xerox Star offers a number of facilities developed from the original SMALLTALK concepts. The system offers multi-fount and multi-point text and graphics facilities, all running in a WIMP environment

COURTESY OF RANK XEROX

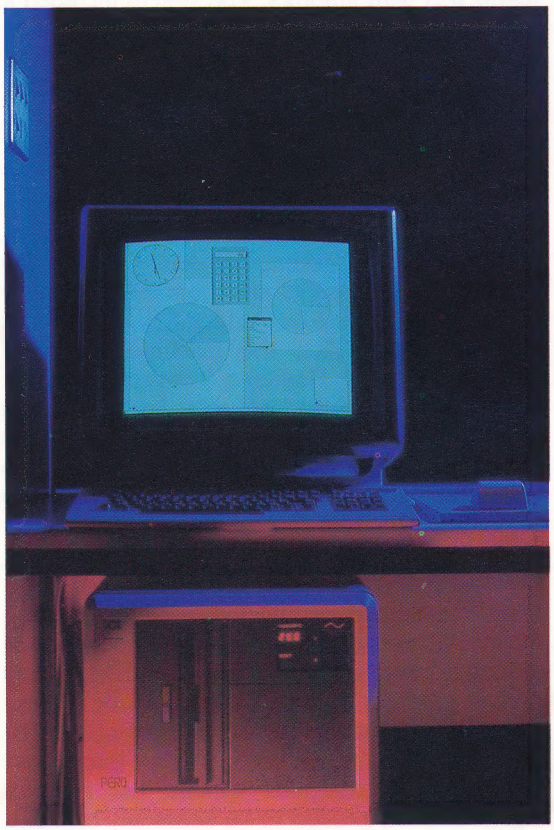
BIT BLOCK TRANSFER

Although modern chip design techniques have contributed greatly to the ease with which such peripheral chips can be implemented, the design of the algorithms themselves also has its origins in the early work done by Xerox. The SMALLTALK algorithm was named BITBLT (for 'BIT BLock Transfer'). It is sometimes colloquially referred to as 'bit-blit' or 'blitter'. In manipulating images, BITBLT does not constantly plot and re-plot them on the screen by using drawing algorithms to alter screen memory. Instead, it operates on the screen memory directly and independently of the main processor, scanning the memory locations in much the same way as an electron beam scans the screen in a VDU.

BITBLT achieves its high speed not only by scanning memory in this way, but also by limiting itself to byte manipulation rather than manipulating screen images at bit level. This may seem like a serious limitation — for example, how do we alter individual bit-mapped pixels if we can only manipulate them in groups of eight? And when we wish to move an image across a static background, how do we restore the previously overwritten image?

Blitters solve both these problems in one stroke by using high-speed bit-wise logic operations — ANDing, ORing and XORing screen memory. The algorithm can generate a series of byte masks that are overlaid onto the existing display. These logic operations are extremely fast, and XOR in particular has the advantage of being able to restore a previously overwritten background.

The combination of bit block transfer technology, fast 16/32-bit processors like the Motorola 68000, and modern chip design for peripheral devices has set the scene for affordable and efficient WIMP systems.



MARCUS WILSON-SMITH

Graphic Features
The Perq, manufactured by ICL, was the first British computer to use the concepts pioneered by Xerox. The high resolution screen offered outstanding monochrome graphics potential, and the machine was mainly sold to scientific research establishments. One early user was the Edinburgh University artificial intelligence unit, which used the Perq's graphics facilities for (among other things) experiments in robot vision

MICRO CASINO

Having outlined the basic statistical techniques in the last instalment, we now consider some of the ways in which artificial intelligence can be employed to the benefit of the computer-based gambler. First, we look briefly at some of the betting systems that have found favour.

When someone tells you about an infallible gambling system, he should be treated with extreme scepticism. However, there are a few systems that perform well, within the conditions for which they are designed. One of these was invented by d'Alembert, the illegitimate son of a French noblewoman, who ran a celebrated Parisian salon. d'Alembert was a friend of Voltaire and Diderot, and contributed to the *Encyclopedie.* His system requires that the true odds in your favour are evens or better. On the whole, this is unfounded, but if you have a good prediction program it may be true. d'Alembert himself used his staking system to amass a tidy sum in the days when roulette wheels were fair and there was no house advantage.

Nowadays, roulette is worse than useless (unless you find a wheel that is bent in your favour), but if your prediction program is giving better than even payback, this method can increase your gains. It works as follows.

- Start with a stake of 5 units (£5 for example).
- After each win decrease your stake by one unit.
- After each loss increase your stake by one unit.
- If you ever get down to zero, re-start at 5 units.

The method's efficacy is very complex, but it can be demonstrated empirically on a computer. We have written a simulation program to give you an idea of what to expect from such a system. This is quite easy to do since BASIC has a built-in random number generator (the RND function).

You can vary the probability of a win with this program, and use it to see how that affects the expected payoff. You can also adapt it to test other betting strategies of your own devising, without losing a penny. In fact, the great advantage of simulations like these is that they keep you out of the betting shop! They provide test-beds for trying out your wider theories and discarding them without damage to your wallet. Just occasionally they may also confirm that you have hit on a worthwhile plan.

BLACKJACK STRATEGY

The essence of d'Alembert's method is that it magnifies your expected winnings. If the odds are against you, however, it magnifies your losses too. A different strategy can be used in another casino game — blackjack — to turn the house advantage in your favour. You simply bet high on good hands and bet the minimum allowed on bad ones. Naturally, in order to do so, you have to distinguish good from bad hands by becoming what is known in the trade as a 'counter'. That is, you must memorise the cards already played.

All blackjack counting systems descend from that used by Edward Thorp in the early 1960s. He made a fortune at Las Vegas, and wrote a book about his methods called *Beat the Dealer.* Although careful use of his system does swing the odds towards you, it has two major drawbacks:

1. The expected rate of return on turnover is in the region of 0.5 per cent.
2. Casino staff are trained to spot counters.

The first problem derives from the second. Because the system can easily be detected, sophisticated versions of the scheme call for more subtle behaviour, thus limiting potential gains. This means that if you stake £100 per hour you can only expect to win about 40p.

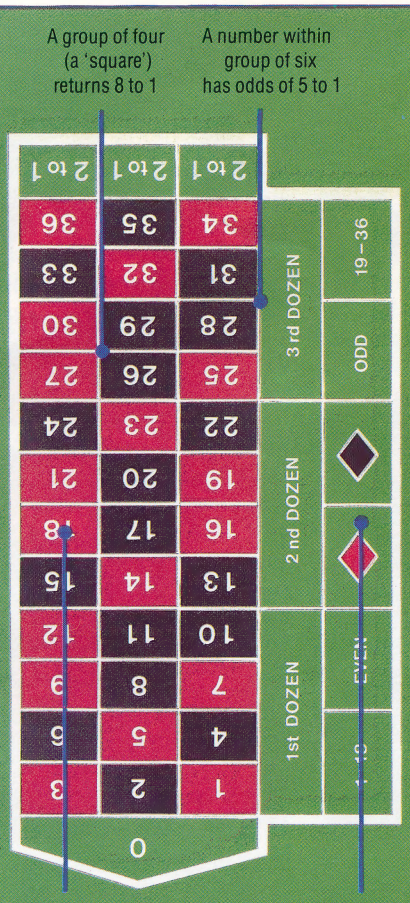
The second problem means that after exhibiting the staking profile of a successful counter for a while, you are likely to be invited by the manager to a free dinner on the house. If you refuse, you will almost certainly find yourself being 'escorted' off the premises by a club bouncer. Let's therefore leave the smoke-filled atmosphere of the casino and move on to an area that has so far been surprisingly nelected: the application of AI techniques to gambling.

IMAGEBANK

Gold Fever
Casinos attract many different types of player, but serious punters remain in a minority. Despite the profit margin enjoyed by the house in all gambling pastimes, some prediction systems have been developed to maximise the punter's returns. Unfortunately, most systems demand a staking pattern that can be easily recognised by the casino staff, and a systematic punter may find himself being politely requested to leave the table.

Out For A Spin

The casino game of roulette is an obvious example of the house weighting mathematical probabilities in its favour. In the American version of the game the table is laid out as shown here and players can place chips at various points to bet on single numbers, groups of numbers or on a particular colour. The true probability of a particular number coming up is 36 to 1 but the house actually gives odds of 35 to 1 allowing itself, on average, a 2.7 per cent profit margin. On even money bets (odd/even, red/black, 1-18/19-36) the house returns half the stake placed if a zero comes up, reducing its profit margin to 1.35 per cent



KEVIN JONES

AI AND THE GAMBLER

One of the most interesting developments to come out of AI in the past decade is the expert system. As we saw in an earlier series (see page 1301), an expert system shares many of the characteristics of a human expert — including the ability to handle uncertain inference in ill-defined situations by means of 'fuzzy logic', or probabilistic reasoning.

When an expert system is being developed for use in prospecting for precious ores or diagnosing illness, the first thing the designers do is enlist the help of a human specialist in the field who can do the job with a high degree of skill. The expert's knowledge is then codified and refined for use by computer. The same approach could be used with, say, horse racing.

A researcher at Carnegie-Mellon University in Pittsburgh, Stephen F. Smith, has developed a learning program called LS-1 that learned to play draw poker.

Draw poker is a game for two or more players. Five cards are dealt to each player. The players in turn have the option of calling, betting or dropping. When the first call is made, each player may replace up to three cards from his hand with new (unseen) cards from the pack. A drop terminates the current round of play. A bet consists of placing some money, at least as much as the previous bet, into a central 'pot'. If a second call is made, all hands are shown and the player holding the hand with higher rank collects all the money in the pot.

LS-1 is a learning system employing an evolutionary or 'genetic' adaption algorithm (see page 1348). Its main loop goes as follows:

1. Generate a starting population of rules at random.
2. Evaluate all the rules and if the average score is good enough, stop and display them.
3. Otherwise, for each rule compute its selection probability p=e/E, where e is its individual score and E the total score of all the rules.
4. Generate the next population by selecting according to the probabilities calculated in 3 and applying certain genetic operators; then repeat from step 2.

Each pass round this loop corresponds to a single generation.

The genetic operators mentioned are crossover, inversion and mutation. They are meant to simulate, in simplified form, what happens in the process of evolution. Crossover is a mating procedure, whereby information from two parental structures are merged to form a new 'offspring', as a candidate rule for testing. Inversion reorders information in the rules. Mutation makes haphazard changes.

Clearly this competition between rule structures has much in common with competition between natural organisms. The rules that have the greatest chance of leaving descendants are those that perform best in the task.

In the case of draw poker, the results were impressive. LS-1 was pitted against a hand-crafted poker program, since no human opponent would have the patience to play it over 40,000 rounds. For every bet decision it had four alternative actions: bet high, bet low, call, or drop. Starting from scratch, with a random initial collection of rules, LS-1 proved so strong that the computer player had to be re-programmed to provide a more formidable opponent. The modified program proved a sterner test, but after several thousand more rounds, LS-1 was able to win nine out of ten hands. By the time it finished, it was one of the strongest non-human poker players in the world.

RACECOURSE STUDY

Another pilot study, also with an evolutionary flavour, was carried out using the BEAGLE learning system (Biological Evolutionary Algorithm Generating Logical Expressions, see page 1321) on a sample of horse racing data from the summer of 1982.

The data covered 153 horses from 51 all-age handicaps with 10 runners or less. Only the top three horses in the betting forecast of each race were considered. The database was split into two parts: 99 records were used as a training set (for forming the rules); and the remaining 54 were used as test data (to see if the rules generalised to fresh examples). Each horse was measured on 18 variables, including the following:

SPEED	speed figure for the horse
RATING	whether highest, next highest speed figure, and so on
FC	position in betting forecast
LAST	place in last outing
LASTONE	place in last but one outing
LASTTWO	place in last outing but two
DAYS	days since last race
SPOTFORM	Daily Mirror Spotform in rank order
BIN	distance beaten in best recent performance
GOING	state of the course
WT	weight carried

In addition, variable WIN was used to record whether the horse won.

The BEAGLE program uses an evolutionary learning scheme similar to that outlined in the previous section (see page 1322). When given the horse racing data it came up with rules such as (SPEED>60) and (WT>(SPEED*2.18)), which between them help to distinguish likely winners from losers. These rules are used jointly to give a 'signature' or 'fingerprint'. That is to say, if rules 1 and 3 are true and rules 2 and 4 are false, the signature would be the binary number 0101 (because the system works backwards) or 5 decimal. This index 5 points to position 5 in a table where information about that particular combination of rule values is accumulated.

When applied to unseen data by the LEAF module (Logical Evaluator And Forecaster), the computer-generated rules were correct 80 per cent of the time. Of course, you can be right 72 per cent of the time, simply by saying 'no', since most horses lose. But an examination of the printout demonstrates that the rules were acting as a very effective filtering system. BEAGLE only predicted four winners: they all won. This is a good sign because selectivity is the essence of scientific betting. (You cannot expect to predict every race; the important thing is to wait until you find a 'racehorse among donkeys' and then strike.) Moreover, its top 13 forecasts contained 10 winners and only three losers. In other words, the nine horses in the 0.4854 probability group did better than expected, with six winners.

Looking at this from another angle, the number of winners out of the bottom 41 was only five. The lowest category, the 'no-hopers', contained only one mistake, so it is good at filtering out the also-rans. (Note that LEAF put question marks in the forecast listing to warn about predictions made on the basis of small samples.)

All in all, then, there is evidence that learning techniques developed for AI research can pay its way on the racecourse. This is only a preliminary study, but the results are highly encouraging. There are a wealth of techniques in AI literature just waiting to be exploited. There are learning algorithms, pattern recognition methods, schemes for handling uncertain inference, and much more besides. We have only scratched the surface here.

Staking Simulator

```
 10 REM ******************************
 15 REM ** D'ALEMBERT'S METHOD:     **
 20 REM ******************************
 75 @%=4
100 PRINT'"Staking simulator:"
101 REPEAT
110   PRINT
120   PRINT "Please give probability of success ";
122   INPUT PS
125   UNTIL PS>0 AND PS<=1
130 INPUT "How many trials ", T
140 N=0
150 GAIN=0: LOSS=0
155 LAST=0: ST=-99
160 W=0
190 REM -- Main Loop:
200 REPEAT
202   N=N+1
210   GOSUB 1000 : REM compute stake
212   PRINT N;"  ";
220   IF RND(1) <= PS THEN S=1 ELSE S=0
230   IF S THEN GOSUB 1500 ELSE GOSUB 1600
240   LAST=S
250   UNTIL N>=T
300 PRINT: PRINT "Total won = ";GAIN-LOSS
330 PRINT "Proportion of successes: ";W/T
360 END
999 :
1000 REM -- Stake Decision Routine:
1010 REM can be altered for experiments:
1020 IF LAST THEN ST=ST-1 ELSE ST=ST+1
1030 IF ST<1 THEN ST=5
1040 RETURN
1050 :
1500 REM -- Success Routine:
1510 GAIN=GAIN+ST
1520 PRINT "Won  ";ST;TAB(15);"Winnings = ";GAIN-LOSS
1525 W=W+1
1550 RETURN
1560 :
1600 REM -- Failure Routine:
1610 LOSS=LOSS+ST
1620 PRINT "Lost ";ST;TAB(15);"Winnings = ";GAIN-LOSS
1650 RETURN
1660 :
```

COMPLETE SHEET

We round up our spreadsheet series with save and load routines that allow us to save complete sheets of data and formulae. We also include a help screen that details which keys activate the spreadsheet functions for each of the four micros.

The final addition to our spreadsheet program is the implementation of the save and load routines, which enable us to create permanent records of data or formulae on disk or tape. Since each of the four machines for which the spreadsheet is designed handle files in different ways, we include separate listings for the Amstrad CPC range, Commodore 64, BBC Micro and Sinclair Spectrum.

In all four versions, the load and save routines start at line 7000 by displaying a short submenu for the various options available. These are:

1 — Load a DATA sheet
2 — Load a FORMULA array
3 — Save a DATA sheet
4 — Load a FORMULA sheet
'X' — To exit

You can see from the options menu that the program allows you to save or load independently data or formulae. This feature is particularly useful since it lets you create a formula template for a particular task (say, calculating VAT returns — see page 1564), save it on tape or disk and then load it up whenever required so that a new set of data can be worked on each time. Conversely, you may have a job that requires several different operations to be done on the same set of data, in which case the original set of data can be stored and reloaded when required for each new calculation.

Implementing this facility for loading and saving individual formulae or pieces of data is actually very easy. You will remember that all the data values for the sheet are held in the two-dimensional numeric array M(,) and the corresponding cell formulae are held in a one-dimensional string array F$(). All we need to do is save the appropriate array as a sequential file. To reload, we simply read the file back into the array from which it was taken, although, as you would expect, the previous contents of the array would be lost after a load.

SEQUENTIAL FILES

Let's start our discussion of how the program saves the arrays by lookiong at the Amstrad CPC and BBC Micro versions. These machines use a similar syntax for opening and closing files. OPENIN opens a file for input and OPENOUT opens a file that is to be written to, but whereas the BBC Micro file-handling system allocates a channel number as part of the file-opening operation, the Amstrad only allows a single channel (#9) to be used for its PRINT and INPUT operations on files.

The save routines merely run through the appropriate array, an element at a time, using a FOR . . . NEXT loop (or in the case of the two-dimensional array M(,), a pair of nested loops) to PRINT them to the sequential file. The load routine is merely the reverse of this process.

Although the Spectrum does not support sequential files, it will allow you to simulate the process by saving numeric or string arrays to tape using the syntax:

```
SAVE I$ DATA F$()
```

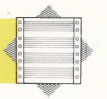
where I$ is the filename and F$() is the array to be saved. Because all we need to do is save arrays, this system is ideally suited to our application.

THE COMMODORE VERSION

Finally, we come to the Commodore 64 version. In developing the program, you may find what appears to be a bug in Commodore BASIC. A sequential file can be set up from a numeric array and read back without any difficulty. The problem arises when writing the elements from a string array to a file.

When a string array is DIMensioned, all the elements are initially set to null strings, represented in the Commodore system by CHR$(0). If we imagine using the spreadsheet to program in some formulae, but leaving some cells without formulae, the corresponding elements in the formula array will remain null and be written to the file as CHR$(0) when F$() is saved. The problem arises when reading the file back using INPUT#.

While this command will read any formulae that were set prior to saving, it will not read back those array elements that were null and the system will hang up. You can get around this 'feature' by testing each string array element before writing it to the file — but if the element was zero, you can write CHR$(1) rather than the CHR$(0) normally written automatically by the Commodore 64. Although CHR$(1) does not represent any character, it gets around the failure of INPUT# to recognise CHR$(0) and so all we need to do is modify the routine that reads the file back into F$(). This means that any elements read as CHR$(1) are set to the null string (see lines 7425 and 7230 of the Commodore version).



The routines that appear in the Commodore 64 listing are for disk systems. For tape systems, make the following changes:

```
7110 GOSUB 7500:OPEN 1,1,0,I$
7219 GOSUB 7500:OPEN 1,1,0,I$
7310 GOSUB 7500:OPEN1,1,1,I$
7410 GOSUB 7500:OPEN 1,1,1,I$
```

This concludes our spreadsheet series. Initially, we outlined some of the shortcomings of our program. For instance, the cell width is fixed at five characters, only numeric data can be entered into a cell and error checking of input has been simplified to shorten the listing. You might, as a further project, like to add to and extend some of the features we've presented.

HELP!

The subroutine at line 6000 provides a HELP screen outlining the keys that must be pressed to obtain the various functions. What it doesn't tell you is exactly what each function does, so let's end the series by briefly running through each spreadsheet function and describing how it works.

ENTER NEW FORMULA IN CELL: A formula can be entered into the cell at the current cursor position. Formulae normally relate constants and cells mathematically (referred to by name, such as A1, B12), and are used to direct the spreadsheet CALCULATE function.

STORE CURRENT SHEET: The data in the sheet can be temporarily set aside within the program. This feature can be used to try out different calculation stategies on the same sheet of data or merely used as insurance against corrupting the data by misdirecting the CALCULATE function.

GET STORED SHEET: This retrieves a previously stored sheet of data.

CALCULATE: This uses the formulae and data entered in the sheet to direct its operations, placing the results of the calculations in appropriate cells.

REPLICATE: It is often useful to be able to reproduce a particular cell formula across a row or down a column. The required syntax is N1 (N2-N3), where N1 is the name of the cell from which the formula is to be replicated, and N2 and N3 are the limits between which the replication is to take place. The routine automatically selects row or column replication.

CLEAR SHEET: Resets the data in every cell to zero.

TAB CURSOR: The spreadsheet cursor can be moved using the cursor keys but jumps to a particular cell can be made using this function. Enter the name of the cell you wish to go to.

LOADING/SAVING ROUTINES: This selects the load/save submenu, which in turn allows you to save the data or formulae currently in the sheet to disk or tape and load previously stored data or formulae.

File Handling & Help Screens

Commodore 64:

```
6000 REM ***** HELP SCREEN *****
6010 PRINT CHR$(147):PRINT
6020 PRINT:PRINT "        F1 - PRINTS THI
S HELP SCREEN "
6030 PRINT:PRINT "        F2 - ENTER NEW
FORMULA IN CELL  "
6040 PRINT:PRINT "        F3 - STORE CURR
ENT SHEET        "
6050 PRINT:PRINT "        F4 - GET STORED
 SHEET           "
6060 PRINT:PRINT "        F5 - CALCULATE
SHEET            "
6070 PRINT:PRINT "        F6 - CLEAR SHEE
T                "
6080 PRINT:PRINT "        F7 - REPLICATE
FORMULA          "
6090 PRINT:PRINT "        F8 - LOADING/SA
VING SHEETS      "
6100 PRINT:PRINT "        'G' - TO TAB CU
RSOR             "
6110 GET A$:IF A$="" THEN 6110
6120 GOSUB 1000:GOSUB 1700:RETURN
7000 REM ***** LOAD/SAVE ROUTINES *****
7010 PRINT CHR$(147):PRINT:PRINT
7020 PRINT:PRINT "          F1 - LOAD DAT
A SHEET"
7030 PRINT:PRINT "          F3 - LOAD FOR
MULA ARRAY "
7040 PRINT:PRINT "          F5 - SAVE DAT
A SHEET "
7050 PRINT:PRINT "          F7 - SAVE FOR
MULA ARRAY"
7055 PRINT:PRINT "          'X' - TO EXIT
"
7056 PRINT:PRINT:PRINT
7060 GET A$:IF A$="" THEN 7060
7070 IF A$=CHR$(133) THEN 7100
7080 IF A$=CHR$(134) THEN 7200
7090 IF A$=CHR$(135) THEN 7300
7095 IF A$=CHR$(136) THEN 7400
7096 IF A$="X" THEN GOSUB 1000:GOSUB 170
0:RETURN
7100 REM **** LOAD DATA ****
7110 GOSUB 7500:I$="0:"+I$+",S,R":OPEN 2
,8,2,I$
7120 FOR I=1 TO 15:FOR J=1 TO 15
7130 INPUT#2,M(I,J)
7140 NEXT J,I
7150 CLOSE2
7160 GOTO 7010
7200 REM **** LOAD FORMULA ARRAY ****
7210 GOSUB 7500:I$="0:"+I$+",S,R":OPEN 2
,8,2,I$
7220 FOR I=1 TO 225
7230 INPUT#2,F$(I):IF F$(I)=CHR$(1)THEN
F$(I)=""
7240 NEXT I
7250 CLOSE2
7260 GOTO 7010
7300 REM **** SAVE DATA ****
7310 GOSUB 7500:I$="0:"+I$+",S,W":OPEN 2
,8,2,I$
7320 FOR I=1 TO 15:FOR J=1 TO 15
7330 PRINT#2,M(I,J)
7340 NEXT J,I
7350 CLOSE2
7360 GOTO 7010
7400 REM **** SAVE FORMULA ARRAY ****
7410 GOSUB 7500:I$="0:"+I$+",S,W":OPEN 2
,8,2,I$
7420 FOR I=1 TO 225
7425 IF F$(I)=""THEN PRINT#2,CHR$(1):GOT
O7440
7430 PRINT#2,F$(I)
7440 NEXT I
7450 CLOSE2
7460 GOTO 7010
7500 INPUT "ENTER FILENAME:";I$
7510 RETURN
```

F1 — Prints HELP screen
F2 — Enter new formula in a cell
F3 — Store current sheet
F4 — Get stored sheet
F5 — Calculate sheet
F6 — Clear sheet
F7 — Replicate formula
F8 — Load/save routines
'G' — Tab cursor

BBC Micro:

'H' — HELP screen
'F' — Enter new formula in cell
'S' — Store current sheet
'G' — Get stored sheet
'C' — Calculate sheet
'Z' — Clear sheet
'R' — Replicate formula
'T' — Tab cursor
'D' — Load/save routines

```
 1189 IF A$="D" THEN GOSUB 7000:REM LOAD
 SAVE ROUTINES
 6000 REM **** HELP SCREEN ******
 6020 CLS:PRINT TAB(6,2)"H - PRINT THIS
HELP SCREEN"
 6030 PRINT TAB(6,4)"F - ENTER NEW FORMU
LA IN CELL"
 6040 PRINT TAB(6,6)"S - STORE CURRENT S
HEET"
 6050 PRINT TAB(6,8)"G - GET STORED SHEE
T"
 6060 PRINT TAB(6,10)"C - CALCULATE SHEE
T"
 6070 PRINT TAB(6,12)"Z - CLEAR SHEET"
 6080 PRINT TAB(6,14)"R - REPLICATE FORM
ULA"
 6090 PRINT TAB(6,16)"T - TAB TO NEW CUR
SOR POS."
 6100 PRINT TAB(6,18)"D - LOADING/SAVING
 ROUTINES"
 6110 A$=GET$:GOSUB 1000:GOSUB 1700:RETU
RN
 7000 REM ***** LOAD / SAVE ROUTINES ***
**
 7010 CLS:PRINT TAB(6,4)"1 - LOAD DATA S
HEET"
 7030 PRINT TAB(6,6)"2 - LOAD FORMULA AR
RAY"
 7040 PRINT TAB(6,8)"3 - SAVE DATA SHEET
"
 7050 PRINT TAB(6,10)"4 - SAVE FORMULA A
RRAY"
 7055 PRINT TAB(6,12)"'X' - TO EXIT"
 7056 A$=GET$:A=VAL(A$)
 7060IF A>0 AND A<5 THEN ON A GOTO 7100,
7200,7300,7400
 7096 IF A$="X" THEN GOSUB 1000:GOSUB 17
00:RETURN
 7100 REM ***** LOAD DATA ******
 7110 GOSUB 7500:F=OPENIN I$
 7120 FOR I=1 TO 15:FOR J=1 TO 15:INPUT#
F,M(I,J):NEXT J,I
 7130 CLOSE#F:GOTO 7010
 7200 REM **** LOAD FORMULAE ****
 7210 GOSUB 7500:F=OPENIN I$
 7220 FOR I=1 TO 225:INPUT#F,F$(I):NEXT
 I
 7230 CLOSE#F:GOTO 7010
 7300 REM **** SAVE DATA ****
 7310 GOSUB 7500:F=OPENOUT I$
 7320 FOR I=1 TO 15:FOR J=1 TO 15:PRINT#
F,M(I,J):NEXT J,I
 7330 CLOSE#F:GOTO 7010
 7400 REM **** SAVE FORMULAE ****
 7410 GOSUB 7500:F=OPENOUT I$
 7420 FOR I=1 TO 225:INPUT#F,F$(I):NEXT
 I
 7430 CLOSE#F:GOTO 7010
 7500 INPUT TAB(0,22)"FILENAME"I$
 7510 RETURN
```

Amstrad CPC 464/664:

'H' — HELP screen
'F' — Enter new formula in cell
'S' — Store current sheet
'G' — Get stored sheet
'C' — Calculate sheet
'Z' — Clear sheet
'R' — Replicate formula
'T' — Tab cursor
'D' — Load/save routines
Note: Press CAPS LOCK before running

```
1175 IF A$="Z" THEN GOSUB 5000:REM CLEAR
 SHEET
1177 IF A$="S" THEN GOSUB 5150:REM STORE
 SHEET
1178 IF A$="H" THEN GOSUB 6000:REM HELP
SCREEN
1189 IF A$="D" THEN GOSUB 7000:REM LOAD/
SAVE ROUTINES
6000 REM **** help screen ****
6020 CLS:LOCATE 6,3:PRINT"H - Print this
 HELP screen"
6030 PRINT:PRINT TAB(6)"F - Enter new FO
RMULA"
6040 PRINT:PRINT TAB(6)"S - STORE curren
t sheet"
6050 PRINT:PRINT TAB(6)"V - Get PREVIOUS
 sheet"
6060 PRINT:PRINT TAB(6)"C - CALCULATE sh
eet"
6070 PRINT:PRINT TAB(6)"Z - CLEAR sheet"
6080 PRINT:PRINT TAB(6)"R - REPLICATE fo
rmula"
6090 PRINT:PRINT TAB(6)"G - GO to new ce
ll"
6100 PRINT:PRINT TAB(6)"D - SAVE/LOAD ro
utines"
6110 WHILE INKEY$="": WEND
6120 GOSUB 1000:GOSUB 1700:RETURN
7000 REM **** LOAD/SAVE ROUTINES ****
7010 CLS:LOCATE 6,4:PRINT"1 - LOAD DATA
SHEET"
7020 PRINT:PRINT TAB(6)"2 - LOAD FORMULA
 ARRAY"
7030 PRINT:PRINT TAB(6)"3 - SAVE DATA AR
RAY"
7040 PRINT:PRINT TAB(6)"4 - SAVE FORMULA
 ARRAY"
7050 PRINT:PRINT TAB(6)"'X' - TO EXIT"
7055 A$="":WHILE A$="":A$=INKEY$:WEND
7056 A=VAL(A$)
7060 ON A GOTO 7100,7200,7300,7400
7096 IF A$="X" THEN GOSUB 1000:GOSUB 170
0:RETURN
7100 REM **** LOAD DATA ****
7110 GOSUB 7500:OPENIN I$
7120 FOR I=1 TO 15:FOR J=1 TO 15
7130 INPUT#9,M(I,J)
7140 NEXT J,I
7150 CLOSEIN:GOTO 7010
7200 REM **** LOAD FORMULAE ****
7210 GOSUB 7500:OPENIN I$
7220 FOR I=1 TO 225
7230 INPUT#9,F$(I)
7240 NEXT I
7250 CLOSEIN:GOTO 7010
7300 REM **** SAVE DATA ****
7310 GOSUB 7500:OPENOUT I$
7320 FOR I=1 TO 15:FOR J=1 TO 15
7330 PRINT#9,M(I,J)
7340 NEXT J,I
7350 CLOSEOUT:GOTO 7010
7400 REM **** SAVE FORMULAE ****
7410 GOSUB 7500:OPENOUT I$
7420 FOR I=1 TO 225
7430 PRINT#9,F$(I)
7440 NEXT I
7450 CLOSEOUT:GOTO 7010
7500 LOCATE 1,22:INPUT"FILENAME";I$
7510 RETURN
```

Sinclair Spectrum:

'H' — Prints HELP screen
'E' — Enter numeric data
'F' — Enter new formula
'C' — Calculate sheet
'S' — Store current sheet
'G' — Get stored sheet
'Z' — Clear sheet
'R' — Replicate formula
'T' — Tab cursor
'D' — Load/save routines
Note: Press CAPS LOCK before running

```
6000>REM PRINT HELP SCREEN
6010 CLS
6020 PRINT : PRINT "  H - PRINTS
 THIS HELP SCREEN"
6030 PRINT : PRINT "  E - ENTER
NUMERIC DATA"
6040 PRINT : PRINT "  F - ENTER
NEW FORMULA"
6050 PRINT : PRINT "  C - CALCUL
ATE SHEET"
6060 PRINT : PRINT "  S - STORE
CURRENT SHEET"
6070 PRINT : PRINT "  G - GET CU
RRENT SHEET"
6080 PRINT : PRINT "  Z - CLEAR
SHEET"
6090 PRINT : PRINT "  R - REPLIC
ATE FORMULAE"
6100 PRINT : PRINT "  T - TAB TO
 NEW CELL"
6110 PRINT : PRINT "  D - LOAD/S
AVE OPTIONS"
6120 LET A$=INKEY$: IF A$="" THE
N  GO TO 6120
6130 GO SUB 1000: GO SUB 1700: G
O TO 1100
7000 REM LOAD/SAVE OPTIONS
7010 CLS : PRINT
7020 PRINT "    LOADING AND SAVIN
G OPTIONS"
7030 PRINT
7040 PRINT : PRINT "       1 - LO
AD DATA SHEET"
7045 PRINT : PRINT "       2 - LO
AD FORMULA ARRAY"
7050 PRINT : PRINT "       3 - SA
VE DATA SHEET"
7055 PRINT : PRINT "       4 - SA
VE FORMULA ARRAY"
7056 PRINT : PRINT "       X - TO
 EXIT"
7060 LET A$=INKEY$: IF A$="" THE
N  GO TO 7060
7070 IF A$="1" THEN  GO TO 7100
7080 IF A$="2" THEN  GO TO 7200
7090 IF A$="3" THEN  GO TO 7300
7095 IF A$="4" THEN  GO TO 7400
7096 IF A$="X" THEN  GO SUB 1000
: GO SUB 1700: RETURN
7100 REM **** LOAD DATA *****
7110 GO SUB 7500
7120 LOAD I$ DATA M()
7130 GO TO 7010
7200  REM **** LOAD FORMULAE ***
*
7210 GO SUB 7500
7220 LOAD I$ DATA F$()
7230 GO TO 7010
7300 REM ***** SAVE DATA ***
7310 GO SUB 7500
7320 SAVE I$ DATA M()
7330 GO TO 7010
7400 GO SUB 7500
7420 SAVE I$ DATA F$()
7430 GO TO 7010
7500 PRINT AT 18,0
7510 INPUT "ENTER FILENAME";I$
7520 RETURN
```

STRING

This is a one-dimensional character array of varying length. *Strings* are one of the main tools at the disposal of the programmer and string manipulation is a major facet of computing.

Perhaps the most common use of the string is for inputting commands to a program. The characters contained in a string can be compared with strings already held in memory and Boolean conditions applied to them, which result in a decision. Alternatively, strings may be combined, or individual characters or groups of characters can be removed from a string and manipulated separately.

STRUCTURED PROGRAMMING

An approach that has become increasingly popular, *structured programming* is a method of program design that involves modular construction. A structured program is first divided into sections covering major tasks and then further subdivided into discrete modules, which perform the simpler tasks while remaining integral to the program as a whole. Because tasks are performed entirely within one module, or passed to submodules, they can be tested independently.

The features that characterise a structured program are global and local variables (see page 688) and block structures such as REPEAT. . . UNTIL and WHILE. . . WEND. The block structures are self-contained operations whose control and conditions can be governed entirely within the module.

Control is exercised through the natural sequence of a program, with branching allowed either when conditions are met or as a result of iteration. Thus each part of the program has its own control structure.

Structured programming makes debugging and amendment of the program much easier. By following the program flow through its subroutines and procedures, you can tell where a program is going wrong and pinpoint the errors. Furthermore, if extra features are required, these may be added simply by including additional routines.

These rules can be extrapolated to more general programming techniques. Because a structured program has a tree-like structure, with the topmost routines controlling the general flow and the bottom layers processing data, its possible to view the top layers as the 'program design'. Hence, we can build the structure of the program first and worry about its individual components later.

The need for structured programming has arisen largely because of the vast number of programs now in use, resulting in an increased number of data processing staff being employed merely to service and update them.

SUBROUTINE

Subroutines are parts of a program that are called outside the program's main control procedure. Once the subroutine has finished executing, control is transferred back to the instruction after the subroutine call. This is done in BASIC with the use of a RETURN statement.

Subroutines are useful in structured programming, where they are used to form characteristic modules. They also have the advantage of saving memory space, as the same subroutine may be called many times from anywhere within the program. Thus, code does not need to be duplicated.

SUBSCRIPT

The address of a particular element in an array is referred to as the *subscript*. For example, an element within the one-dimensional array can be accessed by the subscript s, hence A(s). Often, an array will have more than one dimension, in which case extra subscripts need to be added.

SUPERCOMPUTER

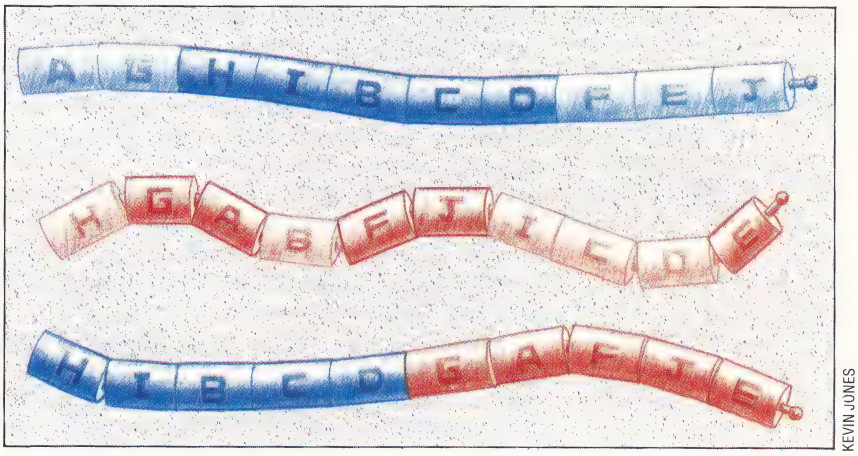
A *supercomputer* is generally regarded as one that can perform over 10 megaflops (a megaflop being a million floating-point operations) per second. Of course, supercomputers are not cheap. The price of one of these machines can run into millions of dollars and only the very largest corporations and defence departments can afford to buy them.

Typically, a supercomputer uses a number of processors running in parallel, as opposed to the single CPU used on most home micros. These machines will run many times faster than other computers: the CRAY-1 ,for example, has a clock speed of around 80MHz, compared with a speed of between 1 and 4MHz on a typical home micro.

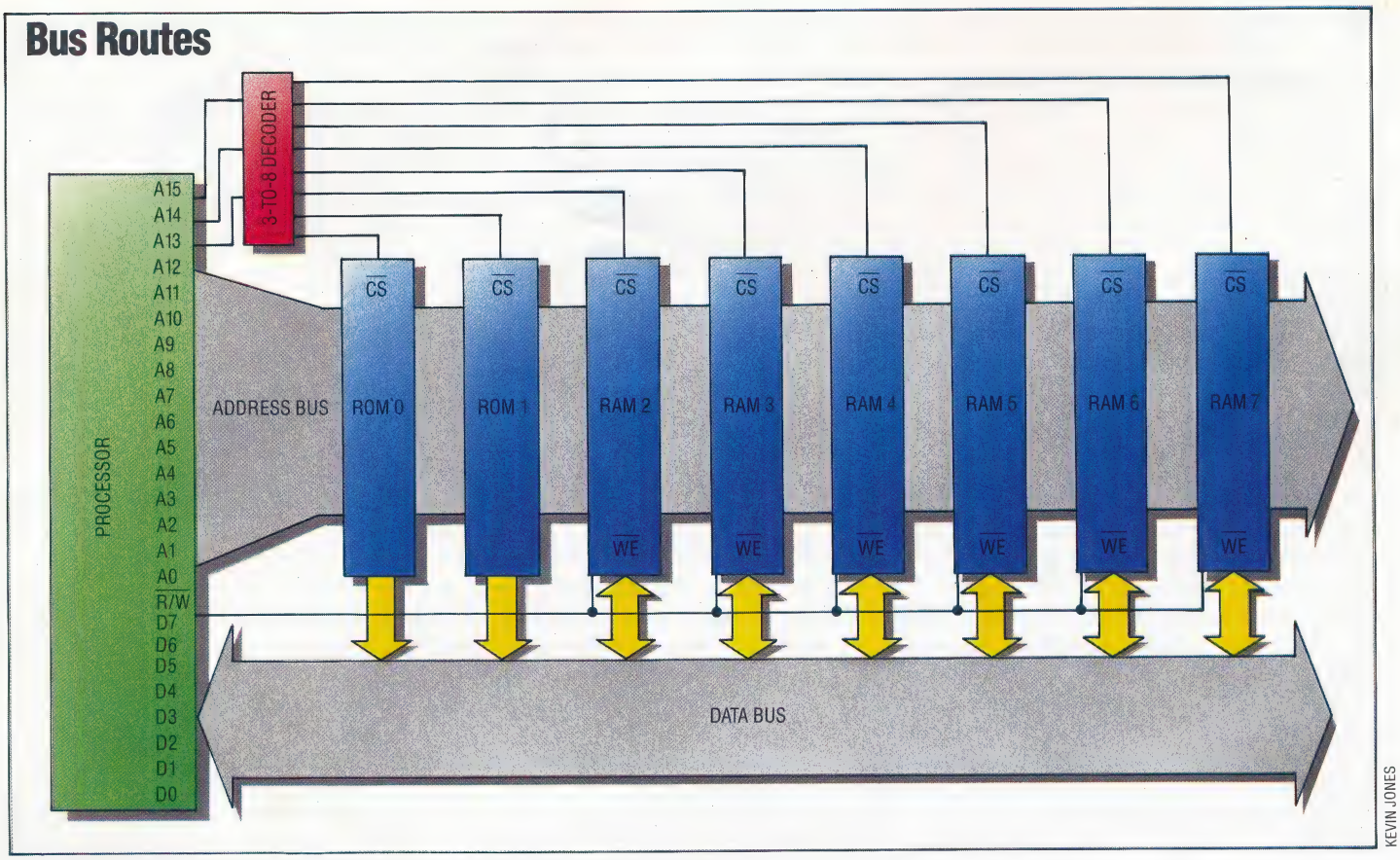
Recently, there has been a further advance in supercomputer technology with the arrival of the first CRAY-2 computer. This machine is notable for the fact that the wiring inside the computer is constantly bathed in liquid in order to reduce the possibility of overheating, thus minimising the risk of system failure or error.

With their enormous number-crunching capabilities, supercomputers are used where large quantities of data need to be processed very quickly. These include applications such as weather forecasting, picture processing and missile trajectory predictions.

String Tuning
Computer languages normally handle textual data as strings of characters and include functions that allow strings to be manipulated and compared. In BASIC, the commands LEN, VAL, STR$, LEFT$, MID$, RIGHT$ and + are used to convert string and numeric data types, take portions of a string and concatenate shorter strings to make longer ones



KEVIN JONES



Lines Of Writing
Here is a typical arrangement of a processor linked into 64 Kbytes of static RAM and ROM. The R/W line from the processor connects to the write enable line on each of the RAMs. When the $\overline{WE}$ line is sent low by the processor, data can be written into the addressed RAM location. The highest three bits are simply decoded into eight chip select lines and are used to enable a particular RAM or ROM. The low 13 bits are bussed to all eight memory chips and specify the location in question

MEMORIES ARE MADE OF THIS

Having looked at the microprocessor itself and the way the control unit co-ordinates the internal actions of the processor, we turn our attention to the memory chips. In particular, we look at how a memory byte may be written to or read.

One piece of computer hardware knowledge that most people understand is the difference between RAM and ROM memory. RAM is memory that can be written to or read from and ROM can only be read from, its contents having been defined during the manufacturing process. However, there are also two main types of RAM, which are known as static and dynamic.

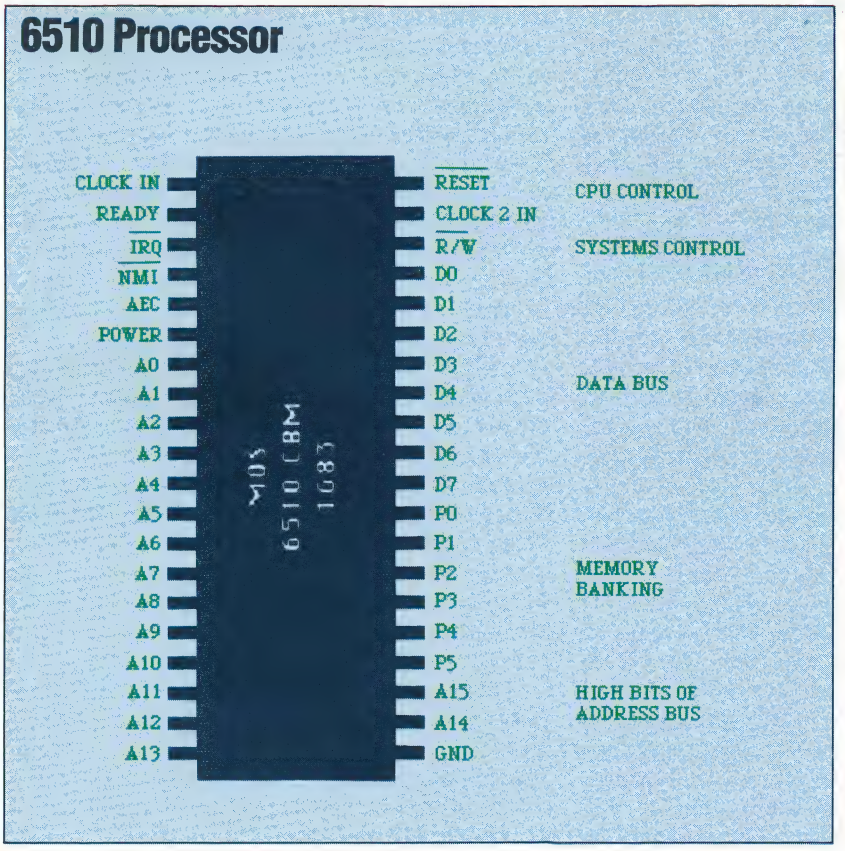
In this instalment, we shall look at how the microprocessor selects a particular memory byte to perform a read or write operation. Let's start by briefly outlining the differences between static and dynamic RAM.

Static RAM uses a small sequential logic circuit known as a J-K flip-flop to store individual bits of data. A flip-flop (see page 608) has the ability to hold a particular state (either zero or one) on its output(s) until a pulse causes it to flip into the opposite state. Eight such devices are used to form each byte of memory and a static RAM chip will therefore have many such devices packed onto it.

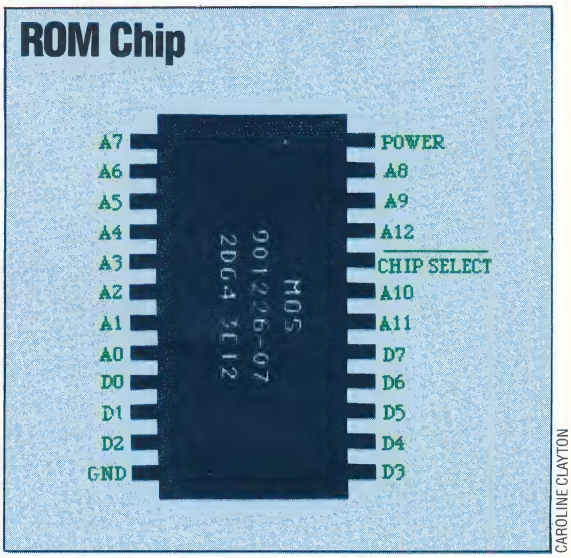
Dynamic RAM stores bits of data as electrical charges. Although storing data in this way takes up much less space (allowing dynamic RAMs to pack more memory per chip), the RAMs have to be refreshed regularly as the charge has a tendency to leak away. We shall be taking a more detailed look at these types of memory in another instalment.

The first diagram shows the pin connections for the 6510 microprocessor (a derivative of the 6502) used by the Commodore 64 and a typical eight Kbyte ROM. The processor has four main pin groups — a 16-bit address bus (A0 to A15), an eight-bit data bus (D0 to D7), CPU control and system control lines. The 6510 has the unusual feature of a six-bit on-chip port that allows, among other things, extra ROM or RAM to banked in and out of its 64 Kbyte address space.

The eight Kbyte ROM shown has 13 address lines (A0 to A12), enabling it to select 8,192 individual locations, eight data lines and a chip select pin. This arrangement is also typical of a

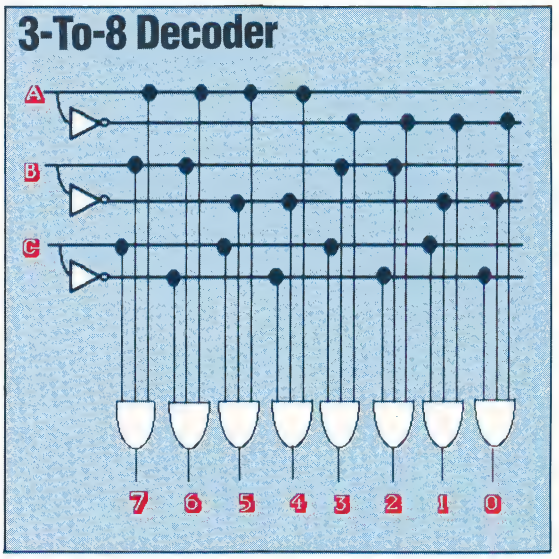


Pinned Down
Most eight-bit processors are designed around a 40-pin format, like the 6510 processor shown here. Notice the 16 address pins, the eight data pins, external clock and control signals. Included in the system control are interrupt inputs and a read/write enable line. The 6510 processor (used in the Commodore 64) is an enhanced version of the 6502, the principal addition being an on-chip data register used for cassette I/O and for controlling the memory banking of the 84K of RAM and ROM



CAROLINE CLAYTON

Select Residence
The 8K ROM chip shown here has a typical pin arrangement for this type of device. Thirteen address lines are needed to address individually each of the 8,192 locations, eight data pins allow direct connection to the data bus and a chip select line is used in conjunction with an external decoder to select that ROM rather than others in the system



Cracking The Code
Decoder circuits have many applications in computer hardware. Their job is to accept a coded binary input and output individual signals that correspond to each permutation of the code. Thus, three inputs can be decoded into eight separate outputs; line 0 corresponds to 000 on the inputs, line 1 corresponds to 001 and so on up to line 7 which is activated by 111 on the input lines

static RAM chip, except that such a RAM chip would have an additional write enable line. The key to addressing ROM or static RAM is the chip select line.

An eight-bit processor has the ability to address up to 64 Kbytes of memory using its 16-bit address bus (as $2^{16} = 65,536 = 64K$). Thus eight separate eight-Kbyte ROM or RAM chips could be accessed by the processor. The address lines A0 to A12 are needed to locate the byte on the memory chip in question, but this leaves A13, A14 and A15, which can be used to select the chip in question. The arrangement of, say, two eight-Kbyte ROMs and six eight-Kbyte static RAMs might be as shown here. The three highest bits of the address are put through a 3-to-8 decoder to select the chip which contained the byte addressed, and the other 13 bits of the address are wired in common to all 8 chips, to select the actual byte on the enabled chip. Thus all locations whose addresses start with 000 can be found on chip 0; locations with addresses starting 001 are selected from chip 1, and so on.

Although ROMs can only be read, RAMs may be read from or written to. To select the operation required, all RAMs have a write enable line ($\overline{WE}$) connected to the read/write line (R/$\overline{W}$) on the processor. In the case of the 6510 the R/$\overline{W}$ line is always held high, except when a write operation is to be made, when it goes low. It's now possible to trace the action of read or write cycles as they affect memory.

When the processor begins to execute a READ instruction, such as 'load the accumulator with the contents of location $C000', the address ($C000) is put on the address bus and the R/$\overline{W}$ line is held high. The upper three bits of address in this example will be 110, so the sixth RAM chip (which is called RAM 6) in our memory arrangement will be enabled via the 3-to-8 decoder. The remaining 13 bits are all zero, so the very first location on RAM 6 will be selected and connected to the data bus so that its contents can be transmitted back to the processor and placed into the accumulator.

During a WRITE instruction (such as 'store the contents of the accumulator in $C000') the location is selected in the same way but this time the R/$\overline{W}$ is held low, causing the write enable pin ($\overline{WE}$) to go low. The contents of the accumulator will already have been placed on the data bus and are written into the first location of RAM 6 at this stage of the instruction.

In the next instalment we'll look a little more closely at the internal structure of memory chips and see how dynamic RAM memory is organised.

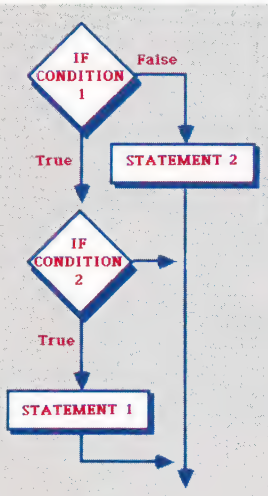
OPERATIONAL DIVISION

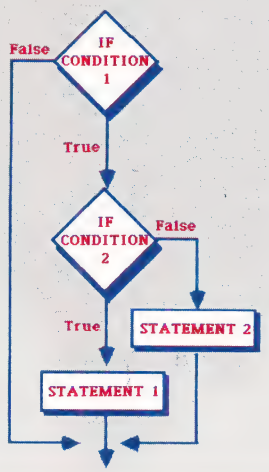
In the previous instalment we looked at how a COBOL program is divided into four parts, known as 'divisions', and considered the functions of the identification, environment and data divisions. It now remains for us to discuss the procedure division — the part of a COBOL program that performs operations and calculations.

The procedure division of a COBOL program corresponds to the program's basic function. The layout of the procedure division and the vocabulary used is meant to make the program as close as possible to business English. Each section is composed of paragraphs, which in turn are composed of sentences that must end in a full stop. Commonly, a sentence would correspond to a statement in a language like BASIC, but a sentence may be further split into clauses (commas are optional), each of which may be a 'statement' or may qualify another 'statement'.

The keyword in a clause or sentence that specifies the action to be taken is referred to as a 'verb'. The most widely used verb by far is MOVE, which simply transfers data from one area to another, taking the form:

```
MOVE identifier-1 TO identifier-2.
```





Trouble With IFs

The use of the full stop as a terminator for the IF construction in COBOL can cause problems. For example, the construction:

```
IF condition1 (then)
   IF condition2 (then)
      statement1
ELSE condition 2.
```

should execute as shown in the top flow chart. However, the role of the full stop as terminator of BOTH statements means that COBOL compiles the unwanted alternative shown below.

The way round this problem is to put the inner selection in a separate paragraph and then to PERFORM that paragraph. So the above construction is best coded in COBOL as:

```
IF condition 1 (then)
   PERFORM paragraph1
   ELSE statement2
.....
paragraph1
   IF condition2
      statement1
```

Even such a simple idea as this has considerable ramifications in a language like COBOL. To begin with, there are two types of move: the alphanumeric move simply transfers character by character from the left, truncating or adding blanks as necessary; the numeric move positions the decimal point properly and performs any changes of usage that are necessary. The numeric move also truncates or pads with zeros either before or after the point. The type of move depends on the PICTUREs (see page 1663) of the two data items.

The fun really begins when you consider MOVEs between alphanumeric and numeric items — some MOVEs are simply not allowed under any circumstances, but these mainly involve the use of special editing PICTUREs and alphabetic-numeric MOVEs. While there is a detailed specification of what happens in every legitimate MOVE, it's nevertheless complicated, so for the moment it's sufficient to say that a statement that easily allows you to truncate significant digits with no run-time message is one to be used with care.

Arithmetic is not COBOL's strong point, though it must be said that few languages give you COBOL's 18-digit precision as a standard feature. Arithmetic can be done in two ways. First, there are the arithmetic verbs ADD, SUBTRACT, MULTIPLY and DIVIDE. For example:

```
ADD number-1 TO number-2.
```

adds the contents of number-1 to the contents of number-2, and:

```
ADD number-1 TO number-2 GIVING number-3.
```

forms the sum of number-1 and number-2 in number-3. There is a further option that allows you to carry out the same arithmetic in a number of places:

```
ADD 5 TO number-1, number-2
```

for example, adds 5 to the current values in both number-1 and number-2.

The other arithmetic verbs follow the same general format:

```
SUBTRACT number-1 FROM number-2 [GIVING
   number-3].
MULTIPLY number-1 BY number-2 [GIVING
   number-3].
DIVIDE number-1 INTO number-2
   [GIVING number-3]
   [REMAINDER number-4].
```

In each case, note that the final value is in the last-named data item.

As with MOVEs, trouble occurs when the result of an arithmetic operation is too large for the space set aside to store it. If least significant digits are lost, they would normally be truncated, except when the name of the receiving data item is followed by the word ROUNDED. Truncation of the most significant digits is, of course, much more serious. COBOL will allow this to happen but it does give you the option of adding an ON SIZE ERROR clause to any arithmetic statement. This has no effect

unless truncation of most significant digits would occur, in which case it leaves the previous contents intact and performs the action specified in the clause.

As you can imagine, anything other than the simplest arithmetic can get very long-winded in COBOL. Matters have been simplified by the introduction of the COMPUTE verb, which can be followed by an arithmetic statement of a form similiar to a BASIC or FORTRAN assignment statement, such as:

COMPUTE number-1 = (number-2 + 3) * number-3.

Some compilers even allow the use of exponentiation within such an expression, but this isn't standard.

There are several problems arising from the use of COMPUTE, one of which is the problem of SIZE ERRORS that may occur on any intermediate results as well as the final result. An ON SIDE ERROR clause may be used but it will not tell you at which stage in the calculation the overflow occurred. For this and other reasons, many COBOL purists do not use the COMPUTE verb at all, but rely on the ordinary arithmetic verbs, which may be long-winded but can be much clearer and safer.

The procedure division is set out in a number of paragraphs. Each paragraph is named; the name is written in the A area starting in column 7 and the statements are written in the B area, starting in column 12. These paragraphs operate in much the same way as a BASIC subroutine. They are therefore not usually separate modules or procedures as in PASCAL and there are no parameters or local variables. Normal execution of the code is from the start through to the finish, executing each paragraph in turn as it is encountered.

The sequence of control in COBOL can be

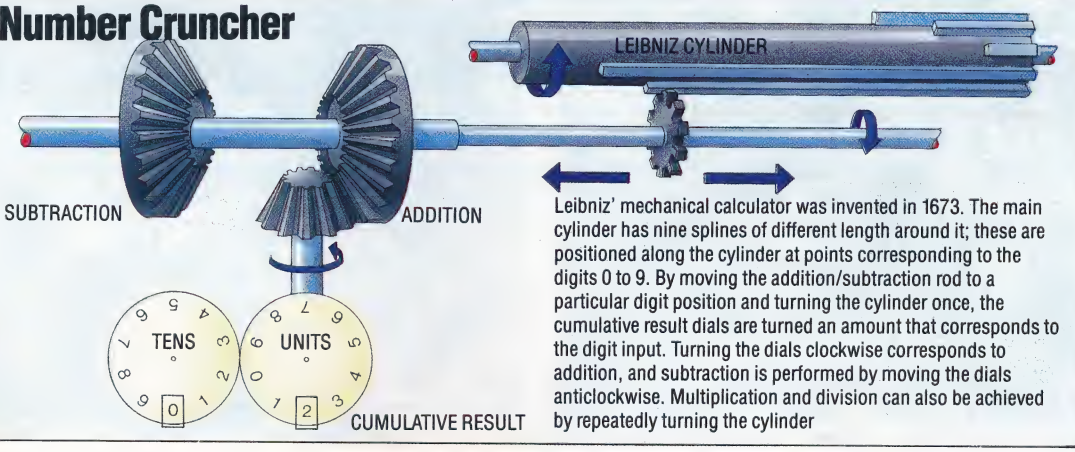
```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. PI-CALC.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.
 OBJECT-COMPUTER.
 SPECIAL-NAMES. CONSOLE IS CRT.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01  SCREEN PIC X(1920).
*
 01  DI-1 REDEFINES SCREEN.
     02 FILLER PIC X(160).
     02 DI-TX1 PIC X(160).
     02 DI-TX2 PIC X(13).
     02 DI-TERM PIC X(15).
     02 FILLER PIC X(136).
     02 DI-TX3 PIC X(6).
     02 DI-PI PIC X(15).
     02 FILLER PIC X(1415).
*
 01  DI-2 REDEFINES SCREEN.
     02 FILLER PIC X(333).
     02 DI-TERM2 PIC X(15).
     02 FILLER PIC X(142).
     02 DI-PI2 PIC X(15).
     02 FILLER PIC X(1415).
*
 01  WORK-AREA.
     02 PI PIC S9V9(14).
     02 TERM PIC S9V9(14).
     02 W PIC S9V9(14).
     02 N PIC 9999.
     02 N1 PIC 9999.
     02 N2 PIC 9999.
     02 ED PIC -9.9(12).
*
 01  CONSTANTS.
     02 TX1 PIC X(17) VALUE "CALCULATION OF PI".
     02 TX2 PIC X(12) VALUE "NEXT TERM IS".
     02 TX3 PIC X(5) VALUE "PI IS".
*
 PROCEDURE DIVISION.
 LA-START.
     DISPLAY SPACE.
     MOVE SPACE TO SCREEN.
     MOVE TX1 TO DI-TX1.
     MOVE TX2 TO DI-TX2.
     MOVE TX3 TO DI-TX3.
     MOVE 0.5 TO ED.
     MOVE ED TO DI-TERM.
     MOVE 3 TO ED.
     MOVE ED TO DI-PI.
     DISPLAY DI-1.
     MOVE 0.5 TO PI.
     MOVE 0.5 TO TERM.
     MOVE 3 TO N.

LOOP.
     MOVE N TO N2.
     SUBTRACT 2 FROM N2.
     MULTIPLY N2 BY N2.
     MULTIPLY N2 BY TERM.
     MOVE N TO N1.
     SUBTRACT 1 FROM N1.
     MULTIPLY N BY N1.
     MULTIPLY 4 BY N1.
     DIVIDE N1 INTO TERM.
     IF TERM < 0.0000000000001 THEN GO TO HALT.
     ADD TERM TO PI.
     MOVE PI TO W.
     MULTIPLY 6 BY W.
     MOVE W TO ED.
     MOVE ED TO DI-PI2.
     MOVE TERM TO ED.
     MOVE ED TO DI-TERM2.
     DISPLAY DI-2.
     ADD 2 TO N.
     IF N < 100 GO TO LOOP.
HALT.
     STOP RUN.
```

Calculating Pi
This COBOL program illustrates the use of the arithmetic verbs to calculate a value for PI using the Leibniz series. This algebraic expansion series operates on the principle of calculating an infinite series of terms, with each term of the series becoming successively smaller.

Much of Leibniz's work has relevance to modern computing procedures and, even in his own time, led to the development of 'calculating machines', one element of which is illustrated below. The COBOL program, running on a rather more up-to-date machine, was written under CP/M using MicroFocus CIS-COBOL

Number Cruncher



Leibniz' mechanical calculator was invented in 1673. The main cylinder has nine splines of different length around it; these are positioned along the cylinder at points corresponding to the digits 0 to 9. By moving the addition/subtraction rod to a particular digit position and turning the cylinder once, the cumulative result dials are turned an amount that corresponds to the digit input. Turning the dials clockwise corresponds to addition, and subtraction is performed by moving the dials anticlockwise. Multiplication and division can also be achieved by repeatedly turning the cylinder

KEVIN JONES

changed in a number of ways, chiefly by the PERFORM verb, which has many variations. Two of these are:

```
PERFORM paragraph-1.
```

which acts as a procedure call or a GOSUB, executing the statements within the paragraph and then returning control to the statement following the PERFORM; and

```
PERFORM paragraph-1 THRU paragraph-n.
```

which will execute a number of consecutive paragraphs before returning control.

Another way of affecting the flow of control is to use:

```
GOTO paragraph-1.
```

which works in the usual way, and which should be avoided just as much in COBOL as in other languages.

INPUT/OUTPUT

As a language primarily concerned with business applications, COBOL provides a comprehensive set of read and write options for files on disk or tape. Standard COBOL, however, makes little provision for interactive input/output using a screen and keyboard. The two verbs to handle this are:

```
ACCEPT data-name.
```

for input from the keyboard, and:

```
DISPLAY data-name.
```

These work in a normal teletype mode much like BASIC INPUT and PRINT. Most micro versions of COBOL include enhanced ACCEPT and DISPLAY verbs to cater for full screen work.

One of the main features that is looked for in any modern programming language is the range of control structures that are available, particularly the facilities for selection and iteration. COBOL provides an IF . . . THEN . . . ELSE construction for selection, though it is restricted compared to that offered by, for example, PASCAL. The format is:

```
IF condition
   statement(s)
ELSE
   statement(s).
```

Note that there is no THEN and also that a full stop terminates the IF statement, as it does with any other statement, which means that there can be no full stops among the statements in either the IF part or the ELSE part. This does not matter normally as statements can follow one after another in a single sentence, but it does make a difference if any of the statements offer alternatives, such as another IF or a READ statement that checks for an end of file marker. In these cases, the full stop that terminates the inner selection will also terminate the outer one.

The Boolean conditions tested by IF can be formed in a number of ways. First, they can be set out (as in most other languages) using the relational operators <, > and =, as well as using NOT to create the others, as in NOT < which means the same as >=. The operators can be written as symbols or written out in full, as in number-1 IS EQUAL TO 5. Any number of these conditions can be conjoined with AND and OR to produce a compound condition, as in (number-1 = 5) OR (number-2 NOT = 6).

COBOL is one of the few languages that allows these compound conditions to be abbreviated where the same data name or the same operator is being used again in a compound condition. The following are both legal COBOL conditions:

```
number-1 < 10 AND > 6
letter-1 = 'A' OR 'B' OR 'C'
```

COBOL does not have a Boolean data type, but it provides a special level 88 data entry that can be associated with any elementary data item to specify a value, range or set of values and a Boolean condition name that can be used wherever a normal condition would be used. For example:

```
77 number-1 PIC 99.
88 number-is-valid VALUES 1,3,24 THRU 67.
```

(Note that some COBOLs will only allow either a range or a list of values, but not both.) The condition name number-is-valid is associated with number-1 simply by being written immediately after it in the data division. Whenever a value is stored in number-1, the value of number-is-valid is adjusted to true or false appropriately and can be directly tested, as in:

```
IF number-is-valid
   PERFORM valid-routine
ELSE
   PERFORM invalid-routine.
```

Iteration (looping) in COBOL can be done in a variety of ways, all of which involve variations in the PERFORM statement. The basic form is:

```
PERFORM paragraph-name UNTIL condition.
```

where the condition can be any combination of relationships or level 88 items, as with IF. This acts like a WHILE loop in BASIC and other languages, with the condition being tested before the paragraph is performed each time. If the condition starts off being true, the paragraph is not performed at all.

The closest equivalent to a BASIC FOR . . . NEXT loop in COBOL is a variation on this use of PERFORM, which allows one or more numeric data items to be incremented from a starting value by a step value. The following statement will perform the code in paragraph-1 five times:

```
PERFORM paragraph-1 VARYING number-1 FROM 1
      BY 1
UNTIL number-1 > 5.
```

Note that the terminating condition can be anything — it does not necessarily have to involve the numeric item.

WORKING WITH WORDS

Word processing packages, which now abound for almost every computer, represent by far the most widespread 'serious' application for home micro users. A short series on some of the more popular packages begins here with an overview of word processing in general.

Word processing is essentially a combination of different operations, some of which are user-dependent and some performed entirely by the computer. These operations are the entry of text into the computer, the creation of a file in RAM or on disk to hold the text (usually in ASCII format), the display of all or part of a file on a VDU in readable form, the manipulation of the data in that file and, finally, the manipulation of the various created files themselves.

It is necessary to spell this out, because any word processor must be judged according to the success with which it performs, or enables the user to perform, these operations. Some word processing programs are, for example, particularly good at displaying text (MacWrite, for example, shows a variety of exotic founts on screen), while others excel at file manipulation. In addition to the performance of these relatively low-level tasks, word processing software must also be judged on the power and extent of the high-level facilities it offers.

CHRIS STEVENS

Real Dedication
The Amstrad PCW 8256 is the latest in the range of low priced products from the company. Packaged as a dedicated word processor and based around the Z80 chip with 256 Kbytes of RAM available, the PCW provides a computer, monitor, integrated disk drive, printer and bundled word processing software for just over £400 (including VAT)

Word processing has now become so familiar that the complex programming it requires is often taken for granted. Furthermore, it is an area of programming that is of fairly recent origin. There are a variety of reasons for this. First, a VDU is essential to display the text, and these have only become standard output devices on computers in the last ten years. Secondly, computers are designed to handle numbers, not text. The introduction of the ASCII code obviously solves this problem, but the code holds 255 different characters, and as such is not at all efficient in its use of memory. A standard typewriter keyboard holds just over 100 characters (counting shifted keys), so it's easy to see that a 255-character code has considerable inbuilt redundancy. For this reason, word processing is not only complex, but also very demanding of memory.

This complexity leads to the problem of text manipulation having to be tackled in many different ways. Word processors come in many different shapes, sizes and forms, ranging from a £6,000 dedicated machine down to a £5.95 program designed to run on a home computer. In this series, we will be examining examples from each main category.

COURTESY OF MICROWRITER LTD

Finger Language
The Microwriter offers a completely different concept in word processing from other machines on the market. Disposing of the standard QWERTY keyboard altogether, individual characters are entered by pressing different combinations of the pads. When the text of a document has been entered, the Microwriter is then connected to a printer to obtain 'hard copy'

DEDICATED WORD PROCESSORS

Dedicated word processors, such as those made by Wang and IBM, have become very popular in the business market. They consist of a display terminal, a keyboard with a number of keys specifically used for word processing functions (a Paste key, for example), and a means of storing text files on hard or floppy disk. With the exception of the Amstrad PCW 8256 (see page 1770), such machines are expensive — often costing over £5,000. Nevertheless they do offer a number of advantages that come from specialisation — purpose-built keyboards, networking facilities and ergonomically designed displays. Word processing involves a lot of staring at the screen, so a wide (80-column plus) and clear display is essential.

Computer Companion
Lap-held portable computers are gradually gaining an increasing share of the market. The advantage of these machines is that the user can perform 'computing on the move'. Most of the lap-held computers, like the Olivetti M10 shown here, that are currently available have a word processing capability held in ROM. Although the word processing facilities and the amount of free memory on these computers is restricted due to the limited power that is available, they are useful tools for letters, memos and so on

By contrast, word processing software falls into many different categories. At the lowest level, some form of text manipulation can be accomplished using a simple screen editor, as provided by most operating systems. You can use this facility to print out short paragraphs in direct mode, but obviously this isn't going to suffice for anything except the very briefest memo.

Further up the scale is the 'text handling program', which enables the user to set up a text file and edit it directly on screen. This is much closer to the concept of a proper word processor, and some text handlers even offer search routines (to locate a particular string) and cut/paste functions, which enable the user to manipulate small blocks of text within a file. A good example of such a program is the TEXT facility offered on the Kyocera range of portable computers, badge-marketed on the Olivetti M10 and the Tandy Model 100. This program offers cut, paste, copy, search and print functions, storing text as a file in CMOS RAM, with a battery back-up so that it will still be there next time the machine is used.

WORD PROCESSING PROGRAMS

Finally, we have the programs that can properly claim the title of word processor. These fall into two categories — RAM-based and disk-based. The former category was really introduced for the benefit of home computer users who lacked disk storage and therefore required a program that loaded wholly into RAM and, because cassette storage is so slow, held the entire document in RAM as well. Only when the document has been edited to a final version is it saved to tape. RAM-based programs are limited in the scope of the facilities they can offer for obvious reasons. In particular, document size is severely limited, and the size of blocks of text that can be shuffled around may also be restricted.

RAM-based programs do, however, have one big advantage over their disk-based relatives — they tend to be much faster in operation, since the different facilities do not rely on numerous disk accesses to function. A good example of a RAM-based word processor is Tasword, which allows the user to create a document of up to 13,000 characters, or just over 2,000 words.

Disk-based programs tend to allow you to work on far larger documents and in theory the document size is limited only by the area of storage available on the data disk itself. The program loads portions of the relevant file as required, though this can often lead to considerable delays, especially if you're at the end of a long document and want to return to the beginning. Typical examples of disk-based systems are WordStar and PerfectWriter. Such programs are often bundled with business systems, but otherwise they tend to be expensive — WordStar 2000, for example, costs £465. One advantage of disk-based programs is that they tend to make greater use of the disk-operating system they run under (CP/M, MS-DOS and so on) and will therefore tend to have greater file compatibility than their home computer equivalents.

In this series we will be looking at a number of different types of word processor and assessing their various strengths and weaknesses. Starting with WordStar, a program that has succeeded in establishing an industry standard by which all other programs are judged, we will go on to look at MacWrite, as an example of a program that excels with its display. We will also be discussing various RAM-based programs, such as Tasword and Mini-Office.

MOTOROLA MASTERPIECE

A Chip Fit For The Future
The 68000 series has given today's micros the processing power of a minicomputer, and manufacturers have been quick to take advantage of it. The machines shown here all use the 68000, and powerful peripheral chips to handle graphics and sound have also been developed to run in conjunction with the chip

The vastly improved addressing capabilities of the 68000 microprocessor can rival the power of many mainframe machines. We begin a series on the 68000 with an overview of those aspects of its architecture that are related to programming.

The 68000 microprocessor has evolved from the highly successful 6800 series of micro products from Motorola, the end point of which was the 6809 microprocessor (see page 518). Motorola followed this up with its high-performance 68000 series, made possible mainly as a result of advances in VLSI (very large scale integration) techniques, which allow more than 100,000 electronic logic elements — such as AND/OR gates, flip-flops and so on — to be fabricated on a single 64-pin chip about two inches in length. This achieves similar addressing capabilities to those of the DEC PDP-11 minicomputer, with all the convenience and power of the instruction set of a mainframe computer — all on a single chip.

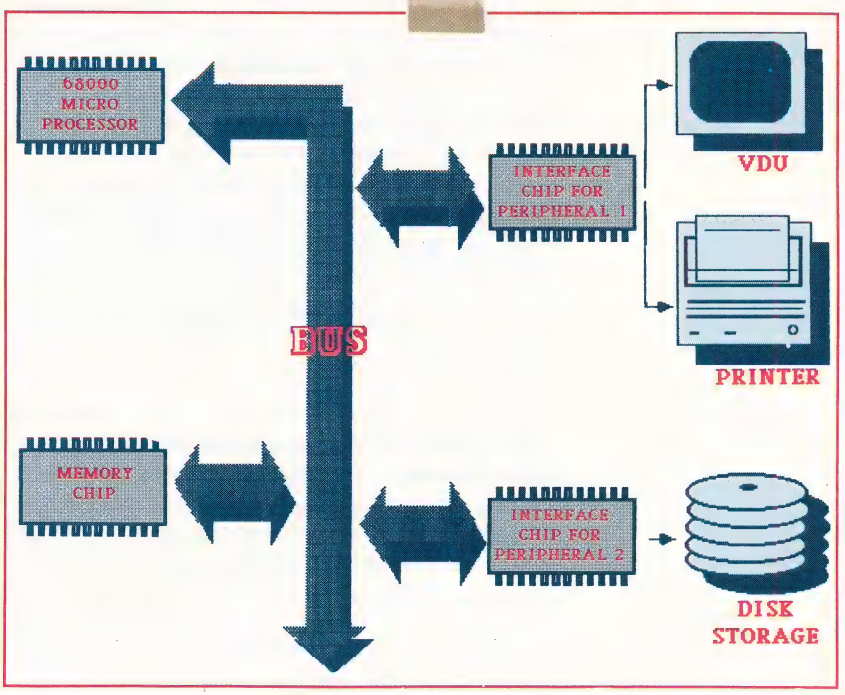
We'll be looking at the addressing and instruction capabilities in later instalments but, for the moment, we'll confine ourselves to the overall architecture of the 68000 so that we have a useful model on which to apply our programs. The idea of a model is important because we need to limit the scope of our knowledge of the microcomputer to those aspects that will allow our programs to function. For example, we do not need to understand the operation of flip-flop elements, in order to add two numbers together and form a result in a register. But we do need to know about the register length and the ADD instructions to solve the problem. Our model is therefore limited to the functions required to program the microcomputer.

THE HARDWARE ARCHITECTURE

If we examine the electrical functions of the 68000 microprocessor pins we see that the signals are concerned with the control of digital buses. These buses are the means by which the processor can communicate with the surrounding electronic world and by which a complete microcomputer system can be built up. A typical system might be built on a single board as shown in the first illustration. Here, the processor can fetch the next instruction from memory via the bus, execute the instruction internally within the processor and, if necessary, send a character to an interface chip to display to the user.

This hardware architecture forms the basis of our model. We still need to think in terms of a

The Basic Model
The diagram shows a simplified representation of a typical single-board computer system. Instructions and data are fetched from memory via the bus and processed. The results are then deposited back on the bus for transmission to the screen, other peripherals, or back into RAM

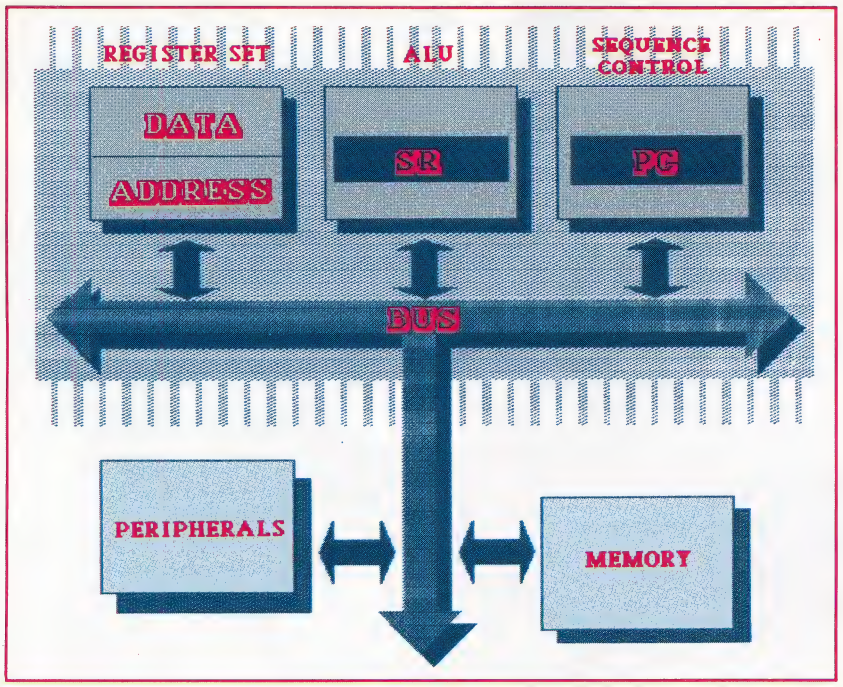


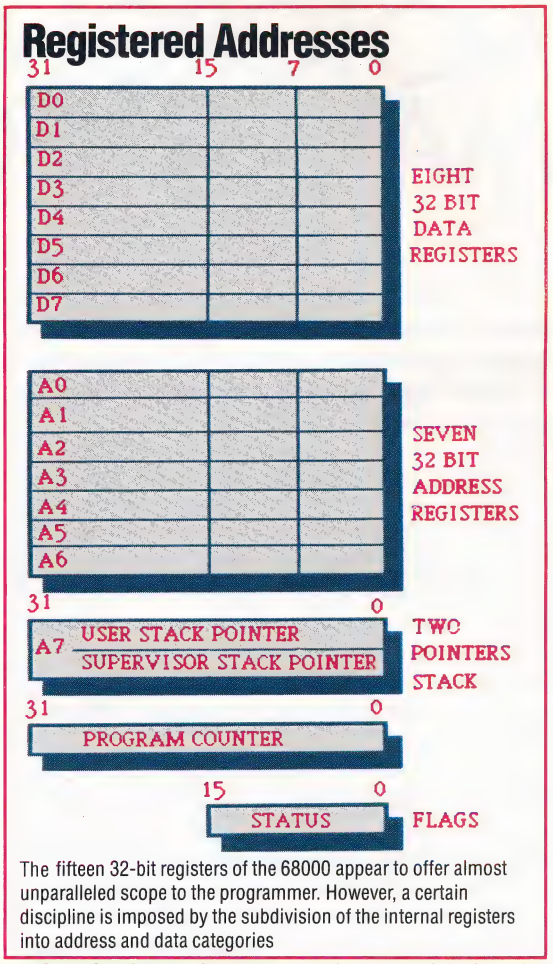
MIKE CLOWES ON THE MACINTOSH

Programmer's Model
The 68000 internal design can be reduced in essence to the provision of three main units: the data and address registers, the arithmetic/logic unit and the sequence control unit. Each unit communicates with the others and with peripheral devices via a digital data bus

central communicating bus but we must also bear in mind the internal register level components of the microprocessor.

The second diagram shows the programmer's model from which we can see that the only difference in hardware architecture is that the 68000 chip has been broken down into three logical components. So let's take a look at each of these in turn.





The fifteen 32-bit registers of the 68000 appear to offer almost unparalleled scope to the programmer. However, a certain discipline is imposed by the subdivision of the internal registers into address and data categories

THE REGISTER SET

These registers are, of course, no more than an array of fast memory locations that are used in the same way as any other computer registers for the temporary storage of data or intermediate results. The difference on the 68000 is that there are eight data registers and eight address registers consisting of 32 bits (in other words, they are 32 bits wide).

Whatever Motorola's reason was for splitting the registers down into two subsets, the result has been to simplify the operations the machine has to carry out and make it very easy to program.

For example, if we wish to load the contents of memory whose address is contained in a register, ready for an arithmetic operation, we know with this architecture that the pointer, or address, must now be loaded into an address register and that the data must be stored in a data register. In assembler language this becomes:

```
MOVE    (A2),D4
```

where MOVE is the data move (or copy) instruction, A2 means take the source address as the contents of address register 2, and D4 means the destination is data register 4. However, if the address of the source data was in D4, we could not use

```
MOVE    (D4),A2
```

as both are illegal source and destination addressing modes with the MOVE instruction.

One further point to note when considering the register sets, and in particular the data registers, is that data to be accessed can be one of five types:

- Bit — a single binary digit.
- BCD digit (or a nybble).
- Byte — an eight-bit character.
- Word — a 16-bit word.
- Long word — a 32-bit word.

We deal with BCD digits and long words in later articles; for the moment we shall consider them as increasingly larger units of data.

Most instructions allow you to specify the data type with the instruction by specifying it as an attribute to the instruction. For example, if we only wanted to copy bits 0 to 7 from D1 to D2 then we would use:

```
MOVE.B    D1,D2
```

but if we wanted the whole 32 bits, then the instruction would become:

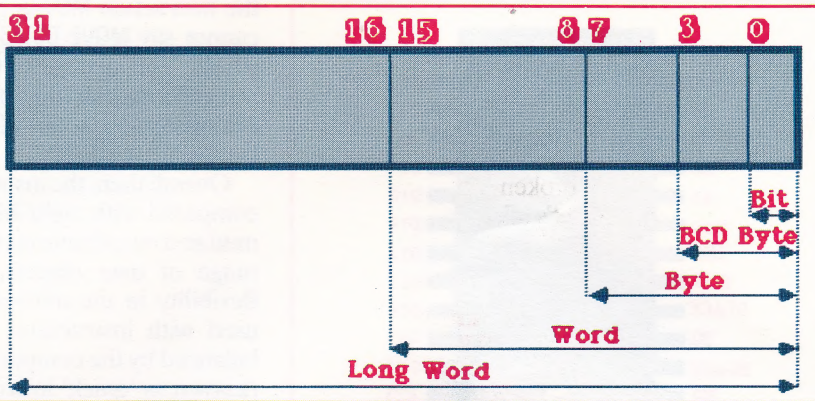
```
MOVE.L    D1,D2
```

The attribute (or data size code) for a 16-bit word is .W, which is the default attribute if none is specified.

A final point before leaving the register set is to note that one of the address registers, A7, is used as a system stack pointer — so we must be careful not to change this register indiscriminately. We shall be looking at stacks later.

Word Lengths

The 68000 can, using extended instructions, operate upon words of five different lengths, processing data in groups ranging in size from one to 32 bits. This offers the advantages of flexibility and more processing power over the older 8-bit chips



THE ALU

The next element to consider in the programmer's model is the arithmetic and logical unit (ALU). This is the part of the micro which actually performs the arithmetic or logical computation and delivers the result to the destination operand. For example, it is the ALU that adds D1 and D2 and places the result in D2 when the following instruction is executed:

```
ADD     D1,D2
```

Again we could specify the data length attribute with this instruction, so:

```
ADD.B   D1,D2
```

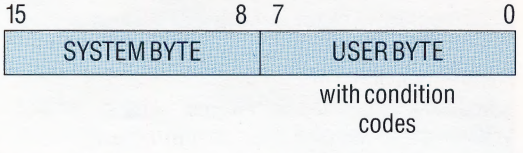
would just add the least significant bytes.

Referring back to the diagram of the programmer's model, there is a register labelled SR, or status register, associated with the ALU. This register is present so that we can remember the result of a previous instruction step. We do this in order to make a conditional branch in the program, depending on the result of the previous instruction execution. For example:

```
ADD     D1,D2    add D1 to D2
BVS     OFLOW    check for overflow
MOVE    D1,D3    copy D1 to D3
```

where we would either branch to the label OFLOW if the overflow bit in SR is set, or MOVE D1 to D3 if it is not. The BVS (Branch if Overflow Set) instruction tests the overflow or V bit in the status register, which could be set as a result of the ADD instruction (as in this particular example). The overflow condition arises, of course, because the result of an arithmetic operation cannot fit into the word size used in the operands. If we do not detect this condition then we will get erroneous answers.

One final point about the status register is that it is 16 bits wide, with parts of each byte being used by the system, as follows:



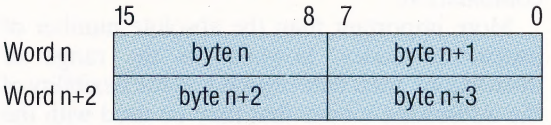
The condition codes are the individual bits set to indicate the result of previous instruction execution — the V bit is an example of one of these codes and will be considered in greater detail later.

SEQUENCE CONTROL

The sequence control section of our microprocessor model contains the program counter, the register that determines the address of the next instruction to be fetched from memory. After an instruction has been fetched, it is decoded by the sequence control to determine which sort of instruction is to be executed by the ALU and where the source and destination operands are.

The program counter is 32 bits wide but the pins through which the microprocessor is connected to the bus only allow 24 bits to be used. Even so, this allows a huge memory address range of up to hex FFFFFF bytes. Each hex digit corresponds to four binary bits, so the 24 bits correspond to a byte address range of 16,777,216. However, notice that all instructions must begin at an even address (or word boundary), so it may be more meaningful to think in terms of a maximum of 8,388,608 words in the address range.

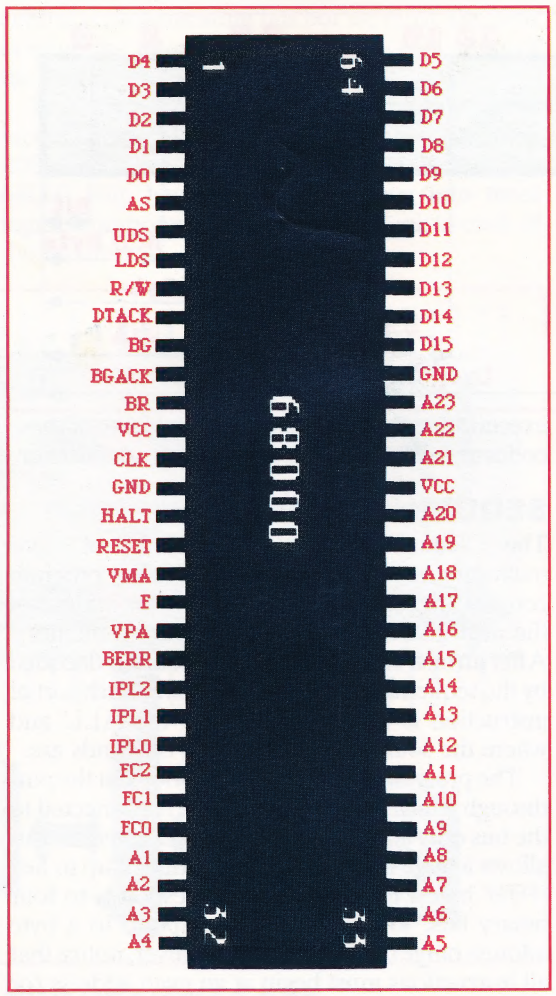
While we are discussing memory access, it is worthwhile considering how data is organised in memory. We need to do this because individual bytes are addressable in this machine, and byte addressing is different to, for example, PDP-11 addressing. The following diagram illustrates 68000 memory addressing showing that the even address gives the most significant byte of the word.



MEMORY ADDRESSING

Some instructions will directly change the program counter in order to cause a program branch — either unconditionally as in BRA BACKHERE (where BRA means 'branch always' to an address represented symbolically by the label BACKHERE), or conditionally as in the previous example, BVS OFLOW, where the branch is conditional on the setting of the V bit in the PS word. In either case, when a branch or a change in the normal sequential execution flow occurs, the program counter will be changed to point to the next instruction to be executed.

Pins-A-Plenty
Shown here are the pin-outs of the 68000. Note the 24 address lines and the 15 data lines. Unlike its close relative, the 68008 (which has only an 8-bit address bus), the 68000 can directly address memory in the range 000000 to FFFFFF hex, or zero to 16,177,216 decimal. Its bigger cousin, the 68020 has a full 32-bit address bus. The 68000 is a perfect example of a modern microprocessor — over 100,000 logic circuits combine to give the processing power of a minicomputer on a single chip



THE INSTRUCTION SET

Computer manufacturers frequently extol the number of separate instructions their computers will support. This can be very confusing as a separate instruction is very often required for each different operand data type or mode of addressing. This is particularly so on older eight-bit microcomputers where, on the 8085, for example, there are 63 different MOV instructions, each of which uses registers in a different combination.

More important than the absolute number of instruction codes, however, is the range of instructions, their data objects and the flexibility of the addressing modes that may be used with the instructions.

Motorola have provided a very powerful instruction set as far as the data objects are concerned, which will allow us to address bytes, words and long words with most of the instructions. We can perform binary multiplication, division and BCD arithmetic, as well as the usual range of logical operations, program control and data copying instructions.

However, although a very useful range of addressing modes is available, we cannot use these without some consideration of the instruction. For example, if the destination operand for a MOVE instruction is an address register then we must use the instruction MOVEA or LEA. In other words we cannot say MOVE D3,A6, but we can say MOVE A D3,A6. The reason is that in providing a very powerful instruction set as far as the data objects are concerned, some of the addressing capability has been sacrificed.

Overall then, the instruction set is very powerful compared with eight-bit micros (and even some mini and mainframe computers), with an excellent range of data objects, but with disappointing flexibility in the addressing modes that may be used with instructions. This is to some extent balanced by the compactness of some instructions (known as 'quick' instructions), where the whole instruction is coded into one machine word. For example, to set a data register to 25 we could use:

```
MOVEQ #25,D3
```

which would move 25 into D3 and the whole instruction would occupy one word, including the data constant of 25.

In the next instalment we will consider in detail the addressing modes available and how we can use them to access data. Some program examples will also be included.

Ahead Of Its Time

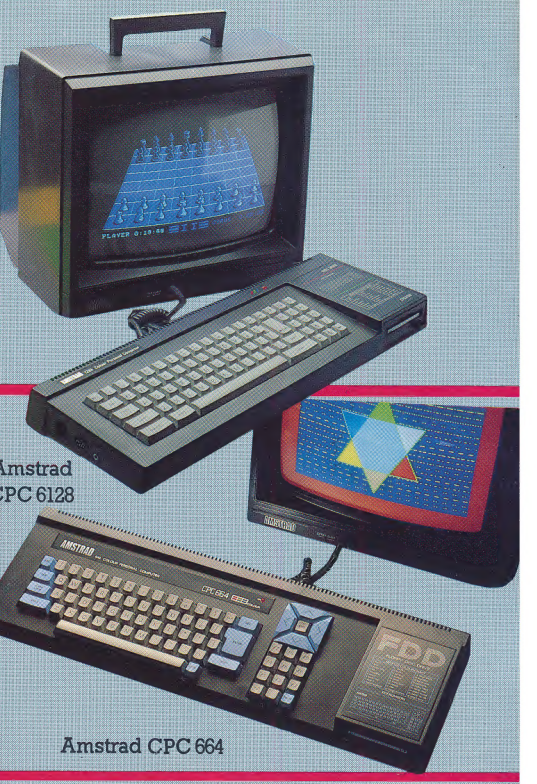
Motorola is a company which has had a chequered history, but nobody can deny that it does design elegant and powerful processors. The eight-bit 6809 arrived too late to make much impact on the explosion in home micros in the late 1970s and early 1980s. The 16-bit 68000, however, was released just as many manufacturers were designing the new generation of machines, and has appeared in various forms on the Sinclair QL, the Apple Macintosh and the Atari ST range.

From the programmer's point of view, the chip is a delight to work with. The processor has 17 32-bit registers, a 16-bit data bus and a 24-bit address bus, and the instruction set.

However, despite the impressive performance of many of the computers that have so far been based around the 68000, there are still problems that must be overcome before it can achieve its full potential. The basic problem so far has been that the 68000 is far in advance of other chip technology, and so manufacturers have been using the chip well beneath its full capacity. An extreme example of this is the Sinclair QL; this computer is based around the 68008 version of the chip, which has only an eight-bit data bus. Thus, for many applications, the QL is not much faster than its eight-bit predecessors.

However, the future of the processor lies with the 68010 chip, which retains the 16-bit address bus but offers virtual memory support. This allows part of the memory to reside in a cheaper back-up storage device, thus enabling it to use far more memory than would otherwise be available. Another Motorola chip — the 68020 — is being hailed as a 32-bit computer on a chip offering 97% of mainframe power!

SOUND SUCCESS



The home micro industry has seen some dramatic developments since The Home Computer Advanced Course began. Many hardware and software manufacturers have felt the pinch from an industry contracting after a period of explosive growth. Companies like Dragon, Oric and Imagine have ceased trading, and even some of the big names – Acorn and Sinclair particularly – have been on the brink of disaster. One company, Amstrad, stands out against this trend. Bobby Pickering asks why.

Amstrad CPC 464

Amstrad CPC 664

The reason why Amstrad, a successful consumer electronics multinational specialising in hi-fi equipment, decided to move into the home micro market is still the subject of speculation. As one story has it, the company found itself at the end of 1983 with a consignment of television monitor components on its hands. What was to be done with them? The solution was simple: replay one of the company's previous marketing triumphs. Amstrad had made its name from developing complete all-in-one hi-fi systems – the so-called 'tower systems' released in 1978. Why not use the monitor components as part of an all-in-one microcomputer system? And so, in April 1984, the company released the CPC 464 on a home micro market dominated by what some regarded as complacent giants – Sinclair, Acorn and, to a lesser extent, Commodore Business Machines .

The CPC 464 was a stroke of marketing genius. In the shops, it physically overshadowed its competitors. The Spectrum looked like a chocolate bar next to it, and compared with the Commodore 64 it offered an inbuilt cassette player and monochrome monitor for a mere £40 more.

Within 18 months, the company could boast a 25 per cent share of the UK home micro market. On top of that, it was increasing its market share at a time when nearly every other micro manufacturer was finding it more difficult to sell its machines.

Amstrad's clear-sighted and sustained marketing strategy can be attributed to one man – its chairman, Alan Sugar, who founded the company in 1968. It started trading under the name Amstrad Consumer Electronics, dealing in the wholesale distribution of car accessories and other finished electrical goods. The company's rocketing growth took off in 1980, shortly after it had conquered the hi-fi market. By that year, Amstrad had grown to become a multinational with sales of some £9 million. Turnover has doubled in each succeeding year, culminating in a figure of £84.9 million in the year ending June 1984.

Before Amstrad came along, the big names in the UK micro industry – Clive Sinclair and Acorn's Curry and Hauser – had reputations primarily based on the part they played in developing and designing their machines. Alan Sugar thus represents a new kind of microcomputer industry personality, since his background is entirely in the consumer market. Amstrad simply assembles electronic components, most of which are imported from the Far East, into consumer goods.

In the short length of time it has been producing computers, Amstrad has certainly made some daring marketing moves. The most remarkable so far came in the spring of 1985. In April the company announced an upgraded 464 – the CPC 664 – which featured a built-in 3in disk drive instead of the cassette player, and included a cutdown version of CP/M. The machine was barely on the shelves, however, when Amstrad announced the CPC 6128, which had twice the memory of a 664 (and a full implementation of CP/M) at a slightly lower price. Although the company had made a major breakthrough in providing a CP/M machine for less than £300, anyone who had rushed out to buy the 664 must have felt cheated. It was almost as if the company was competing against itself.

Amstrad's most recent release is the PCW 8256, announced in August 1985. This is a complete word processor (256 Kbyte micro, monitor, disk drive and dot matrix printer) priced at £399 excluding VAT, which looks destined to fill a gap in the market. A comparable package several years ago would have cost well over £1,000.


© 1983 RUNDGREN, T.